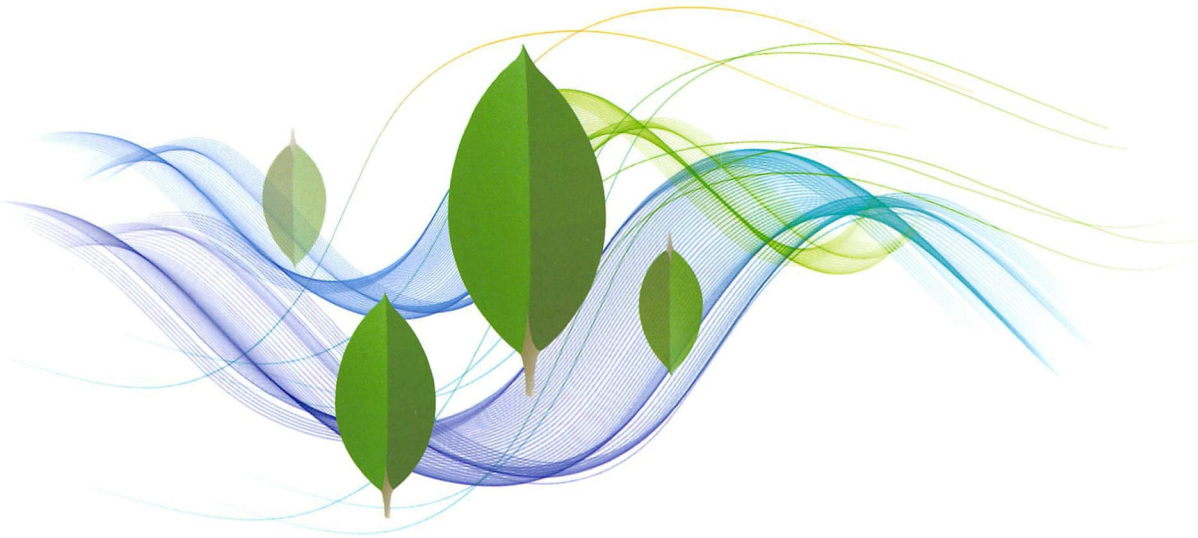
Broadview®
www.broadview.com.cn



MongoDB 运维实战

张甦 贺磊 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

张 甦

数据库专家，51CTO专家博主，《MySQL王者晋级之路》作者。曾就职于某大型电商平台和某汽车门户网站。

十年互联网线上处理及培训经验，专注于MySQL数据库，对MongoDB、Redis等NoSQL数据库及Hadoop生态圈相关技术有深入研究。麾下学员遍布各大企业。

博客：http://blog.51cto.com/sumongodb

贺 磊

目前就职于小米，工作范围包括MySQL和MongoDB平台的架构设计、性能调优、日常运维及自动化开发。知名论坛MySQL版主、51CTO博客之星。闲暇之余，喜欢将部分案例写成博客文章，多篇文章被评为推荐博文、51CTO社区周刊头条，累计访问量过百万。

博客：www.dbapower.com

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

张甦 贺磊 / 著

MongoDB
运维实战

内 容 简 介

MongoDB 自 2009 年推出以来，历经了近十年的发展，在这十年中，数据库领域可谓经历了翻天覆地的变化。

本书深入剖析 MongoDB 新旧版本的特性，结合生产案例详细讲解 MongoDB 的常见故障；引领学习 MongoDB 索引，以便更好地掌握 MongoDB 性能调优技巧；描述备份恢复的重要性，让读者掌握 MongoDB 备份恢复技巧；充分利用 MongoDB 的水平扩展能力，详解 MongoDB 复制集、分片架构环境；最后讲解如何使用 PMM 性能监控平台，做好线上 MongoDB 的监控工作。

图书在版编目（CIP）数据

MongoDB 运维实战 / 张甦，贺磊著. —北京：电子工业出版社，2018.9
ISBN 978-7-121-34989-8

Ⅰ. ①M… Ⅱ. ①张… ②贺… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字（2018）第 207727 号

责任编辑：陈晓猛
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：14.25　字数：273.6 千字
版　　次：2018 年 9 月第 1 版
印　　次：2018 年 9 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐序 1

找到属于你自己的那束光

张甦老师的新书出版，邀我来写几句话，下面谈谈我对数据库领域变革的一点观察和理解。

MongoDB 自 2009 年推出以来，转眼已经近十年，这十年间，正是数据库领域风起云涌的 十年。在同样的时间进程中，阿里巴巴在 2008 年提出去 IOE 理念，推动了中国由互联网至传统行业的数据应用的深刻变革。

我曾经在 2015 年的 DTCC 数据库大会上提出，我们今天已经进入了“后 IOE 时代”，这个时代的典型特征就是“百花齐放”，数据库新产品的不断涌现，为我们带来了新的可能，不同场景可以有多种不同的产品和解决方案，用户因此获得了“自由”。

而 MongoDB 的出现，因其面向文档，具有 Schema Free 等灵活优势，让用户在管理文档、日志，以及基于社交、物流等场景有了一个更好的选择，于是其市场经历了快速的增长，并扛起了 NoSQL 的大旗，也因此在 DB-Engines 的数据库流行度排行榜中，MongoDB 荣膺 2013 和 2014 的年度数据库。

2017 年 10 月，MongoDB 在纳斯达克上市，成为今天市值 30 亿美元的数据库公司，这不得不说是近代数据库历史上的一个巨大成功。而相应地，另外一个更受欢迎的开源数据库 MySQL，几经辗转成为 Oracle 的囊中之物，原因何在？

最近在做一个 MongDB 的迁移，将数据库的存储引擎从 MMAPv1 更换为 WiredTiger。同时回顾了一下历史， 2014 年 MongoDB 收购了 WiredTiger 公司，WiredTiger 为其开发了一个专用版本的存储引擎，今天成为 MongoD 的默认存储引擎，我们不得不钦佩 MongoDB 的英明之处。对比一下 MySQL 的发展历程，当 MySQL 的最佳存储引擎 InnoDB 被 Oracle 釜底抽薪收购（2006 年）之后，MySQL 最后被 SUN 收购（2008 年），辗转落入 Oracle 之手（2009 年），而

自 2009 年 MySQL 5.5 开始， InnoDB 就成为 MySQL 默认存储引擎。

决定一个产品成败的是技术，而决定一家公司成败的，往往是视野。

张甦老师的学习和成长，兼具技术和视野，他不断学习研究和砥砺，使自己获得了深厚的技术体验，而选择 MySQL 和 MongoDB 入行，更可见他对于开源的信心。他此前出版的《MySQL 王者晋级之路》，深受读者欢迎，而 MongoDB 更是指引他向前的“一束光”。现在这束光放射开来，希望能够让更多读者见证 MongoDB 的光彩和未来。

我也祝福走在技术道路上的每一位朋友，能早日找到指引自己的那束光！

盖国强，云和恩墨创始人，Oracle ACE 总监

推荐序 2

近两年，随着互联网的迅猛发展，Oracle 之外的各种开源数据库迅速崛起，数据库领域呈现出了百花齐放、百家争鸣的局面。MongoDB 就是其中最为夺目的一朵开源之花。在 DB-Engines 公布的 2018 年 2 月的数据库排名中，MongoDB 成为榜单中涨幅最大的一个，上涨了 5.47 个百分点，位列榜单第 5 名。

MongoDB 是可以应用于各种规模的企业、各个行业及各类应用程序的开源数据库。MongoDB 能够使企业的数据库更具敏捷性和可扩展性，各种规模的企业都可以通过使用 MongoDB 来创建新的应用。该公司是 NoSQL 数据库技术领域的知名公司。MongoDB 采用分布式基础架构，并且深受移动应用和 Web 应用开发者的欢迎。此外，MongoDB 还是一个基于文件的数据库。在 MongoDB 中，数据被编码成能够兼容多种不同数据格式的文件。MongoDB 的流行程度是显而易见的，目前其应用在全球范围内的下载次数已经突破了 1000 万次。简单来说，使用 MongoDB 能够提高与客户之间的工作效率，缩短产品上线时间，以及降低企业成本。

MongoDB 是一个基于分布式文件存储的数据库，旨在为企业提供可扩展的高性能数据存储解决方案。其数据库特点是高性能、易部署、易使用、储存方便。首先，它支持的数据结构非常松散，是与 JSON 相类似的 BSON 文档格式，因此可以存储比较复杂的数据类型，且该格式文档比较易读、高效。其次，MongoDB 支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，包括传统数据库的功能如二级索引、完整的查询系统等，而且还支持对数据建立索引。此外，MongoDB 的数据可实现复制和故障恢复，还具有在云端的伸缩性，支持水平的数据库集群延伸。

本书作者张甦先生在开源数据库领域深耕细作多年，技术功底扎实，实战经验丰富，对 MongoDB、MySQL 等数据库都有深入的研究，在各大开源数据库技术论坛、网站上颇有名气。

更为难得的是，张甦先生拥有丰富的数据库培训经验，擅长深入浅出地讲解技术问题，能够用简单明了的语言阐述复杂疑难的技术故障。

在数据库领域从业多年，我见过很多技术牛人，但是很多人仅限于自己牛却不擅长将技术分享给更多的人，他们的价值总是有限的，而且是局限于自身的，只有像张甦先生这样能做技术、能讲技术，还能写技术的人才能将技术的价值无限放大，让更多的人受益。

本书是两位作者多年工作和研究经验的总结，语言简单明了，实战案例丰富。本书对工作中常见的各种故障给出了作者的分析和解决方案，具有非常高的实用价值，无论是刚开始学习 MongoDB 的小白，还是有一定工作经验的老 DBA，都可以从中受益。

张甦先生是一位专业的 DBA，也是一位勤奋的写作者，他的上一本书《MySQL 王者晋级之路》才出版不久，我又收到了《MongoDB 运维实战》这本书的手稿，在繁忙的工作之余，还有这样的勤奋和毅力，真是令人钦佩！希望张甦先生能够写出更多更优秀的技术书籍，帮助更多即将走上 DBA 之路的伙伴。

侯圣文，Oracle ACE 总监、教育专家，恩墨学院院长

自序

张甦自序

我出生在北京，小时候的梦想是成为一名足球运动员，可以驰骋于赛场，为自己喜欢的北京国安队效力。从来没有想过会从事数据库相关的工作，更不会想到今后自己会写书。继《MySQL 王者晋级之路》之后，《MongoDB 运维实战》是我写的第二本书。好朋友跟我开玩笑地说，这可能就是一个最美丽的错误。今后可能会出版更多与技术相关的书，就让这个美丽的错误一直延续下去！

在自己近十年的技术生涯中，需要感谢的人真的太多了。并不是说因为现在做出点小成绩，或是因为出书了，就要开始写感谢的话了。在我状态处于最低谷、最迷茫的阶段，是我的那些贵人、前辈和挚友把一直在黑暗中行走的我拉了出来，他们的帮助就好比是一束光，指引着我看清未来的方向。在这条道路上所经历的各种辛酸，只有你们最能理解我！

能写完这本《MongoDB 运维实战》，首先要感谢我的好兄弟贺磊，我们一拍即合。经历了无数个日日夜夜的编写、修改、再编写，目的就是把多年累积下来的工作经验梳理成完整的知识体系分享给大家，让大家在工作中少走弯路，快速达到自己的预期目标。

我们可能永远成为不了那些大腕、明星，我们有的只是一颗本本分分做人、踏踏实实做事、研究技术的心。最后说一句：雷霆雨露，俱是天恩，感谢老天赐予我们的各种艰辛和磨难。正因为经历了这些，才让我们更加珍惜现在来之不易的幸福，使我们更好地去努力奋斗，争取让自己的家庭更好，让我们的父母和孩子为我们所付出的努力而感到骄傲和自豪（送给所有努力工作的兄弟姐妹们）。

望广大读者多提宝贵意见。更希望大家可以花些时间，认真品味 MongoDB 给我们带来的简单快乐。

贺磊自序

在北京这个大城市，时光飞逝，有 4 个人对我有至关重要的影响，我想在这里对他们表示感谢。

首先感谢我的好友张甦先生，是他邀请我一起撰写这本《MongoDB 运维实战》，如果没有张甦先生，可能这些内容会一直躺在我的个人笔记里，或者零零散散地写在我的博客上，并没有成体系。

我一直以来都崇尚分享，我相信分享能够提升自己，因此一有时间，我就毫无保留地在博客上撰写一些实战案例。但零散的博客和写书完全不同，写书不是博客随笔，要有严谨的思路和错误校验，如果没有张甦先生，我可能没有机会将这些知识以书面的形式分享给大家，在他的帮助下，才得此机遇，圆了出书的梦。

其次要感谢我的爱人李爱璇女士，无论在生活中，还是工作中，她对我都是无条件地支持，让我以饱满的精神状态面对工作。多少个下班后的夜晚，是她默默在背后支持我写书，让我非常感动，真心感谢她。

还要感谢卓汝林先生和潘友飞先生，在我入职小米以来，是他们带着我学习、工作，让我的技能水平有了巨大的进步。

MongoDB 的最新版本已经到了 4.0 版本，而目前市面上的书大多数还停留在 2.6 版本和 3.0 版本。MongoDB 在每个新版本里加入了诸多的新特性，本书结合笔者职业生涯中的众多案例，为您一一列出前因后果和处理办法，也希望对得起“实战”二字。在实战中分析、在案例中学习是本书的要义所在，也希望读者看过此书后能有所收获，这是对笔者最大的慰藉。

前言

随着大数据时代的到来及技术的不断发展，以及互联网 Web 2.0 的兴起，传统的关系型数据库在应付超大规模和高并发的 SNS 类型的 Web 2.0 纯动态网站时已经显得力不从心了，暴露了很多难以解决的问题，而非关系型数据库则由于其本身的特点得到了非常迅速的发展。NoSQL 领域首屈一指的就是“芒果”数据库，即大名鼎鼎的 MongoDB。我是从第一个 GA 版本开始接触 MongoDB 的，它在我最孤独、最寂寞的时候，陪伴着我一路成长。倘若我一直在黑暗中行走，那么 MongoDB 就是那一束光，指引着我未来前进的方向。

写此书的目的

我把 MySQL 和 MongoDB 当作自己的两个“孩子”一样看待，一直想把多年运维数据库的经验分享出来，前不久已经出版了一本 MySQL 的著作《MySQL 王者晋级之路》，收到的读者反馈都不错，读者说书很实用，看完之后收获很大。其实写书的真正目的就是为了让大家可以系统地进行学习，少走一些作者在工作中走过的弯路。我的一些学生和朋友经常和我抱怨：“我们公司有一个项目准备用 MongoDB，但还得从头学习，网上针对 MongoDB 的资料也不多，关键还不知道从何学起，MongoDB 实战的书籍也偏少”。这类问题不止一个人和我说过。正因为有了这样的需求，我们才有了想投入全部精力、认真地去写一本有关 MongoDB 实战方面图书的冲动。希望《MongoDB 运维实战》能够真正地帮助大家解决在学习 MongoDB 数据库过程中的诸多疑惑，敲开大数据运维的门。

如何阅读本书

第 1、2 章主要介绍 MongoDB 3.4 和 MongoDB 3.6 这两个版本的新特性，以复制集架构和分片架构作为整体切入点。MongoDB 版本更新到 3.X 之后发生了巨变，进入了一个新的时代。在引入 wiredtiger 存储引擎之后，实现了文档级别的锁，提高了并发性。该引擎支持压缩，节约了存储成本，具有更简单高效的高可用架构，维护起来更加轻松。这让我们对 MongoDB 4.X 时代更加期待。

第 3 章是本书中一道亮丽的风景线，是 MongoDB 的实战案例分析部分。详细介绍 oplog 大小引发的从库宕机、副本集延迟突然增大到上万秒、最大连接数限制等问题的处理过程和思路。

第 4 章是 MongoDB 的性能调优部分。从索引角度出发，通过各类索引的使用，包括配合执行计划的查看来梳理性能问题。

第 5 章介绍 MongoDB 备份与恢复，主要以逻辑备份和物理备份两种方式来进行讲解，其中会演练 oplog replay 的过程。

第 6 章是高可用架构集群管理。核心的两个部分就是复制集和分片架构，包括副本集、分片架构原理、成员类型、实战安装部署过程，以及多种实现方式和管理维护架构中遇到的诸多问题，最后还会介绍升级 MongoDB 架构版本的注意事项。

第 7 章介绍 MongoDB 的监控。主要介绍 PMM，它是一款能够监控 MySQL、MongoDB 性能的开源平台。本章会讲解 server 组件和 client 组件及其安装过程。

第 8 章是 MongoDB 的常用命令。本章列举一些我们在 MongoDB 的运维过程中常用的命令，帮助刚接触 NoSQL 领域的读者快速上手生产环境中 MongoDB 的运维工作，为自己的公司和老板排忧解难。

致谢

在《MySQL 王者晋级之路》中已经说了太多感谢的话。这次先要感谢我的好兄弟贺磊，我们一拍即合，经历了多个日夜，坚持了一年的时间完成了这本 MongoDB 的图书。其次，还是要感谢电子工业出版社编辑陈晓猛先生的耐心指导与对我的支持。

这是我出版的第二本数据库相关的书籍了。第一本是关系型数据库方向的图书，本书是 NoSQL 方向的，未来计划出版一本人工智能方向的图书。道行尚浅，还需努力，希望广大读者多提宝贵意见。更希望大家可以花些时间，认真品味 MongoDB 给运维带来的简单快乐。

张翌

目录

第1章 MongoDB 3.4 新特性

我们在学习掌握一项技术的时候，总会在某个时间段遇到学习瓶颈，遇到瓶颈之后，会有一种失去激情不想学习的感觉，研究技术就会存在这样的问题。一成不变的技术知识，时间长了，我们都会觉得枯燥无味、索然无趣。这时就需要新鲜的血液进入我们的神经中枢，激发我们的求知欲。所以第 1、2 章就是给大家“打鸡血”，详解 MongoDB 3.4 和 MongoDB 3.6 的新特性，让大家感受前所未有的新鲜感，开始我们 MongoDB 的学习之旅。

1.1 复制集（Replica Set）

Default Journaling Behavior of majority Write Concern

配置复制集时，增加 writeConcernMajorityJournalDefault 选项，默认为 true，即当指定 WriteConcern 为 majority 时，数据写到大多数节点并且 journal 成功刷盘后，才向客户端确认成功；如果为 false，则数据写到大多数节点的内存时就向客户端确认。

writeConcernMajorityJournalDefault 参数的具体的默认值与 protocolVersion 相关，当 protocolVersion =1 时，writeConcernMajorityJournalDefault 默认为 true；当 protocolVersion =0 时，writeConcernMajorityJournalDefault 默认为 false。

protocolVersion 从 3.2 版本起默认为 1，老版本的 MongoDB protocolVersion 默认为 0。protocolVersion 为 0 的复制集成员不能加入 protocolVersion=1 的副本集。

MongoDB 3.6 版本已经弃用了 protocolVersion=0。

已弃用的 protocolVersion =0 在低于 3.6 版本的 MongoDB 中都可以配置 protocolVersion=0。

protocolVersion =1 在 MongoDB 3.2 版本或更高版本中可用，并且是默认值。

Adjustable Catchup Period for Newly Elected Primary

配置复制集时，增加 catchUpTimeoutMillis 选项，默认为 2s（3.4.6 版本后调整为 60s，3.6 版本为–1，不限制时间），来指定新选举出来的 Primary 从其他拥有更新数据的节点追数据的时间，增加该时间能最大限度地减少需要 rollback 的数据，但有可能增加整个 failover 的时间。

刚刚选举出来的新的 Primary 节点在完成同其他节点数据同步期间，客户端不能向其写入数据，可以使用{ replSetAbortPrimaryCatchUp: 1 }命令来强制其终止在成为主节点前与其他节点的同步，直接让自己成为 Primary 节点。

catchUpTimeoutMillis 选项只能在 ProtocolVersion 为 1 时使用。

需要注意的是，要将 3.6 版本中启动的副本集降级到 3.4 版本，将 catchUpTimeoutMillis 从 –1 更改为正数。如果未将此值更改为正数，则会导致运行 3.4 版本的节点既不重新启动，也不加入副本集。

支持 Linearizable Read Concern

“linearizable”Read Concern 级别保证，一定能读到 WriteConcern 为 majority，并且确认时间在读请求开始之前的数据，该级别仅在查询结果只有单个文档的情况下有效。

目前 MongoDB 的所有存储引擎都支持“linearizable”Read Concern 级别。

结合“majority”WriteConcern、“linearizable”Read Concern 使多个线程能够在单个文档上执行读取和写入，就好像单个线程实时执行这些操作一样。也就是说，这些读写的相应时间表是线性化的。

Linearizable 级别会明显比 majority 级别或 local 级别的 Read Concern 慢，使用时要与 maxTimeMS 结合使用。

例如：

```
    db.tablename.find( { _id: 5 } ).readConcern("linearizable").maxTimeMS
(10000)
    db.runCommand( {
        find: "tablename",
        filter: { _id: 5 },
        readConcern: { level: "linearizable" },
        maxTimeMS: 10000
    } )
```

Improved Initial Sync

首先看一下 initial sync 的过程：

步骤 1：T1 时间，从 Primary 同步所有数据库的数据（local 除外），复制时，Mongo 会扫描每个源数据库中的每个集合，并将所有数据插入对应的集合。通过 listDatabases + listCollections + cloneCollection 命令组合完成。假设 T2 时间完成所有操作。

步骤 2：从 Primary 应用[T1-T2]时间段内的所有 oplog，可能部分操作已经包含在步骤 1 中，但由于 oplog 的幂等性，可重复应用。

步骤 3：根据 Primary 各集合的 index 设置，在 Secondary 上为相应集合创建 index（每个集合_id 的 index 已在步骤 1 中完成）。

当初始同步完成时，成员从 StartUp2 转换到 Secondary。

在 3.4 版本中：在复制数据的同时建立所有的索引。在早期版本的 MongoDB 中，在这个阶段只有_id 索引被建立。

在 3.4 版本中：initial sync 在数据复制期间拉取新添加的 oplog 记录。需要确保目标成员机器在 local 数据库中有足够的磁盘空间，以在此数据复制期间临时存储这些 oplog 记录。

在 3.4 版本中：改进了初始同步重试逻辑，使其对网络上的间歇性故障更有弹性。

在 3.4 版本中：为避免潜在的数据损坏，在初始同步过程中如果发现在同步源上重命名了一个集合，则初始同步将失败并重新启动初始同步。使用 MongoDB 3.2.11 版本或更低的版本，初始同步不会失败重新启动，而是继续进行，这可能导致潜在的数据被损坏。

在版本 3.4 中：rs.status 命令新增了一些可参考监控项，例如，replSetGetStatus.optimes.lastCommittedOpTime 、replSetGetStatus.optimes.appliedOpTime 、replSetGetStatus.optimes.durableOpTime 等。

```
"optimes" : {
        "lastCommittedOpTime" : {
                "ts" : Timestamp(1527242317, 1),
                "t" : NumberLong(1)
        },
        "appliedOpTime" : {
                "ts" : Timestamp(1527242333, 1),
                "t" : NumberLong(1)
        },
        "durableOpTime" : {
                "ts" : Timestamp(1527242333, 1),
                "t" : NumberLong(1)
        }
```

replSetGetStatus.optimes.lastCommittedOpTime：

从该成员的角度来看，已写入大多数副本集成员的最新操作时间。

replSetGetStatus.optimes.appliedOpTime：

从该成员的角度来看，已应用于副本集的该成员的最新操作时间。

replSetGetStatus.optimes.durableOpTime：

从该成员的角度来看，已写入该副本集的该成员的 journal 日志的时间。

Decimal Type

MongoDB 3.4 新增对 decimal128 format 的支持，最多支持 34 位小数位。

跟 Double 类型不同，decimal 数据存储的是实际的数据，无精度问题。以 9.99 为例，decimal NumberDecimal("9.99") 的值就是 9.99，而 Double 类型的 9.99 则是一个大概值 9.990000000000000213162 8…

例如：

db.inventory.insert({_id: 1, item: "The Scream", price: NumberDecimal("9.99"), quantity: 4 })

为了支持这种格式，MongoDB 在 3.4 版本中新增了 NumberDecimal 参数。

mongo Shell 默认将所有数字视为 64 位浮点双精度值。mongo Shell 提供了 NumberDecimal() 构造函数来明确指定 128 位基于十进制的浮点值，能够以精确的精度仿真十进制舍入。此功能适用于处理货币数据的应用程序，比如财务、税务和科学计算。

十进制 BSON 类型使用 IEEE 754 decimal128 浮点数编码格式，该格式支持 34 位十进制数字（即有效数字）和–6143 到+6144 的指数范围。

NumberDecimal()构造函数接收小数值作为字符串。

NumberDecimal("1000.55") 库中也是这样存储的。

NumberDecimal()构造函数也接收来自 mongo Shell（即不带引号）的 double 值，但由于存在丢失精度的风险，建议不要这样做。该构造函数创建基于二进制双精度表示形式的和基于十进制的参数（可能失去精度），然后将该值转换为精度为 15 位的十进制值。以下示例将该值作为 double 隐式传递，并显示如何以 15 位数的精度创建它：

NumberDecimal(1000.55)

库中存储为：

NumberDecimal("1000.55000000000")

以下示例将该值作为 double 隐式传递，并显示如何发生精度损失：

NumberDecimal(9999999.4999999999)

库中存储为：

NumberDecimal("9999999.50000000")

Collation and Case-Insensitive Indexes

从 MongoDB 3.4 开始支持 collation。在之前的版本中，文档中存储的字符串不论是中文还是英文，不论大小写，一律按字节来对比。引入 collation 后，支持对字符串的内容进行解读，可以按使用的 locale 进行对比，也支持对比时忽略大小写。

下表中的操作支持 collation。

Commands	mongo Shell Methods
create	db.createCollection()
	db.createView()
createIndexes	db.collection.createIndex()
aggregate	db.collection.aggregate()
distinct	db.collection.distinct()
findAndModify	db.collection.findAndModify()
	db.collection.findOneAndDelete()
	db.collection.findOneAndReplace()
	db.collection.findOneAndUpdate()
find	cursor.collation()
mapReduce	db.collection.mapReduce()
delete	db.collection.deleteOne()
	db.collection.deleteMany()
	db.collection.remove()
update	db.collection.update()
	db.collection.updateOne(),
	db.collection.updateMany(),
	db.collection.replaceOne()
shardCollection	sh.shardCollection()

安全提升（Security Enhancement）

3.4 版本支持不停服开启认证，在早期的版本中，如果想将副本集或 sharding 从非认证模式转换为认证模式，则需要停库停服，利用 Transition to Auth 可以实现。

Transition to Auth 允许 mongod 或 mongos 接受使用用户名+密码的登录，也接受未使用用户名+密码的登录，这可以用于副本集或分片集群从 no-auth 配置到开启认证的滚动转换，实现 0 down time 认证切换。该过程要求指定一个内部认证机制，例如 security.keyFile。

例如，如果使用密钥文件进行内部身份验证，则 mongod 或 mongos 会使用匹配的密钥文件与集群中的任何 mongod 或 mongos 创建经过身份验证的连接。如果安全机制不匹配，则 mongod

或 mongos 会使用未经身份验证的连接。

如果在配置文件中开启了 security.transitionToAuth，那么 mongod 或 mongos 将不强制执行用户访问控制，用户可以使用正确的用户名+密码登录或者不使用用户名+密码登录。这部分内容在后面的章节中有单独的实战演示。

工具（MongoDB Tools）

MongoDB 3.4 引入了 mongoreplay 工具，可用于监控和记录 mongod 上执行的命令并“replay”到另一个 mongod 实例上，代替老版本中的 mongosniff。

例如：

```
    [root@HE1 bin]# ./mongoreplay monitor -i eth0 -e 'port 27017'  --collect
json|grep -vE '("request_data":null|ismaster|replSet|"ns":"local")'
    {"order":1,"op":"command","command":"find","ns":"store.$cmd","request_da
ta":{"filter":{"t":{"$gte":150913214822},"ui":10971231239975},"find":"book",
"projection":{"_id":1,"bi":1,"ct":1,"ft":1,"ot":1,"p":1,"s":1,"t":1}},"reply
_data":null,"connection_num":0,"seen":"2018-02-20T14:53:24.280049+08:00","re
quest_id":62612xxxx75}
    {"order":7,"op":"command","command":"find","ns":"store.$cmd","request_da
ta":{"filter":{"c1":{"$in":[{"$numberLong":"337"},-10000]},"fi":307xx2,"ui":
"eb2e3f1cef5d58f9adxxxx375acc1bfc"},"find":"own_fiction_chapter","projection
":{"c1":1}},"reply_data":null,"connection_num":3,"seen":"2018-02-20T14:53:24
.31228+08:00","request_id":139xxxx2420}
    {"order":11,"op":"command","command":"find","ns":"store.$cmd","request_d
ata":{"filter":{"t":{"$gte":151xx720},"ui":"e381955xxxx3803fec57064"},"find"
:"book","projection":{"_id":1,"bi":1,"ct":1,"ft":1,"ot":1,"p":1,"s":1,"t":1}
},"reply_data":null,"connection_num":5,"seen":"2018-02-20T14:53:24.320483+08
:00","request_id":753xxx32}
```

1.2 分片集群（Sharded Cluster）

Membership Awareness

从 MongoDB 3.4 开始，分片集群的所有组件如 shards、config servers、mongos instances 都能在分片集群中互相感知到对方的存在，包括分片集群的名称和 config servers 的位置。这个特性引入了如下限制。

对于 3.4 版本的分片集群，mongod 在实例启动时，必须指定 shardsvr，可以在配置文件中配

置 sharding。clusterRole 为 shardsrv，也可以在启动时加--shardsvr。

配置文件有明确的格式要求，比如：

```
sharding:
  clusterRole: shardsvr
```

具体的分片结构搭建在后面的章节会有详细的讲解。

如果不指定端口，则从 3.4 版本起默认的 shardsvr 成员端口号为 27018。

3.4 版本的 mongos 不能连接低版本的 mongod，要注意兼容性，建议使用同一个大版本的 mongo 客户端来操作 mongod。

Balancer on Config Server Primary

从 MongoDB 3.4 起，Balance 进程从 mongos 移动到了 configer server 的 Primary 节点上。在 MongoDB 3.2 及之前的版本里，分片集群的负载均衡由 mongos 负责，多个 mongos 会抢一个分布式锁，抢锁成功的 mongos 会执行负载均衡任务，在 Shard 间迁移 chunk。

从 3.4 版本起，sh.startBalancer 作为新命令 db.adminCommand({ balancerStart: 1 })的一个包装，执行的时候不会等待 balancing round，而是直接启动，在早期的版本中要等待 balancing round 结束后才可以启动。因此 MongoDB 3.2 或更早版本的 mongo Shell 中 sh.startBalancer()等命令与 3.4 版本的分片群集不兼容。这就是我们强调客户端大版本要与 mongod 一致的问题。例如，使用低版本客户端操作 3.4 版本的分片集群，会出现如下问题：

```
mongos>sh.stopBalancer()
Waiting for active hosts...
Waiting for the balancer lock...
assert.soon failed,msg:Waited too long for lock balancer to unlock
```

这个问题会在后面的章节讲到。

从 3.4 版本起，新增了 db.adminCommand({ balancerStart: 1 })、db.adminCommand ({ balancerStop: 1 })、db.adminCommand({ balancerStatus: 1 })命令，这些命令需在 admin 库下执行，直接使用 sh.startBalancer/sh.stopBalancer 也是一样的效果。

从 3.4 版本起，mongos 的配置文件不再支持 sharding.chunkSize 和 sharding.autoSplit，如果在 3.2 版本的 mongos 里配置了，则在 3.4 版本的 mongos 中要去掉（这在做 sharding 升级的时候要注意）。

MongoDB 3.4 弃用了 mongo Shell 中的 sh.getBalancerHost()命令。3.2 版本或更早版本的 mongo Shell 命令 sh.getBalancerHost()与 3.4 版本分片的集群不兼容。

Faster Balancing

从3.4版本起，WT存储引擎的secondaryThrottle默认值是false（MMAPv1依旧默认为true），chunk migrations 将不再等待操作复制到 Secondary。在实现异步迁移的过程中，如果设置secondaryThrottle 为 true，那么在默认情况下，chunk migrations 迁移期间的每个文档移动会在平衡器继续处理下一个文档之前复制至少一个 Secondary 节点。这相当于{w：2}的 write concern。

从3.4版本起，chunk migration 可以并行操作，在以前的版本中最多只能有一个 chunk 迁移，而从 3.4 版本起，如果有 *n* 个 shard，则可以有最多 *n*/2 个 chunk 同时迁移。例如，sharding 集群有 6 个 shard，那么迁移的时候最多可以同时有 3 个 chunk 进行迁移。

这个 parallel 的数量是可见的。例如，有 6 个 shard 集群，在迁移的时候可以看到有 3 个 chunk 在同时进行迁移：

```
                        shard2  58
                        shard3  58
                        shard4  58
                        shard5  58
                        shard6  58
        too many chunks to print, use verbose if you want to force print
metok_core.geocode_position
        shard key: { "gps_geohash" : 1 }
        unique: false
        balancing: false
        chunks:
                        shard1  399
                        shard2  398
                        shard3  399
                        shard4  19
                        shard5  19
                        shard6  19
        too many chunks to print, use verbose if you want to force print
metok_core.wifi_position
        shard key: { "bssid" : "hashed" }
        unique: false
        balancing: false
        chunks:
                        shard1  2731
                        shard2  2734
                        shard3  2739
                        shard4  49
                        shard5  46
                        shard6  43
```

不再支持 SCCC Config server 的模式

MongoDB 3.2 引入了复制集模式的 config server（CSRS 模式），在此之前，config server 由多个镜像的单节点组成（SCCC 模式）。在 3.4 版本中，MongoDB 不再支持 SCCC 模式的 config server，必须将 config server 部署为副本集架构。

取消 sh.getBalancerHost()命令，因为现在的 Balancer 进程在 config server 的 Primary 节点上。

所以往 3.4 版本升级时，如果 config server 还是 SCCC 模式，则需要先升级为 SCRS 模式。

Sharding Zones

从 3.4 版本起，分片集群里引入了 Zone 的概念，主要取代早期版本的 tag-aware sharding 机制，能将某些数据分配到指定的一个或多个 Shard 上，这个特性将极大地方便 sharding cluster 的跨机房部署，所以我们需要详细了解 Sharding Zones 机制。

为了实现 Zone，可以借助以下命令来实现：

```
sh.addShardToZone()
sh.removeShardFromZone()
sh.updateZoneKeyRange()
sh.removeRangeFromZone()
```

关于 Zone 的知识后面章节会有详细的介绍。

第 2 章 MongoDB 3.6 新特性

Default Bind to Localhost

从 MongoDB 3.6 开始，在默认情况下，MongoDB 二进制文件 mongod 和 mongos 绑定到 localhost（127.0.0.1）上。如果为二进制文件设置了--ipv6 选项或配置文件中配置了 net.ipv6，则会绑定到 IPv6 地址:: 1。

当仅绑定到本地主机时，这些 MongoDB 3.6 二进制文件只能接受来自同一台计算机上运行的客户端（包括 mongo Shell，集群中其他成员的副本集和分片群集）的连接，远程客户端无法连接。

要覆盖并绑定到其他 IP 地址，可以使用 net.bindIp 配置文件进行设置，或使用 bind_ip 命令行选项指定 IP 地址列表。

例如，以下 mongod 实例绑定到本地 localhost 和本地 IP 地址 192.168.1.249：

```
net:
  port: 28000
  maxIncomingConnections: 10240
  bindIp: localhost,192.168.1.249
```

如果连接到此实例，则远程客户端（Host）后跟服务器 IP 地址 192.168.1.249 或与 IP 地址

关联的主机名。

```
[root@HE1 ~]# mongo --host=192.168.1.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://192.168.1.249:28000/
MongoDB server version: 3.6.5
WARNING: shell and server versions do not match
> exit
```

此时使用 mongod 的内网地址 10.10.10.249 是无法连接的，因为 bindip 没有配置。

```
[root@HE1 ~]# mongo --host=10.10.10.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://10.10.10.249:28000/
2018-05-30T19:32:31.883-0700 W NETWORK  [thread1] Failed to connect to
10.10.10.249:28000, in(checking socket for error after poll), reason: Connection
refused
2018-05-30T19:32:31.884-0700 E QUERY    [thread1] Error: couldn't connect
to server 10.10.10.249:28000, connection attempt failed :
connect@src/mongo/shell/mongo.js:240:13
@(connect):1:6
exception: connect failed
```

bind_ip 可以通过 0.0.0.0 来绑定所有，也可以设置 bindIpAll，这样我们可以使用 mongod 服务器的任意 IP 地址来进行登录。

```
net:
  port: 28000
  maxIncomingConnections: 10240
#  bindIp: 0.0.0.0
  bindIpAll: true
```

```
[root@HE1 ~]# mongo --host=192.168.1.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://192.168.1.249:28000/
MongoDB server version: 3.6.5
WARNING: shell and server versions do not match
> exit
bye
[root@HE1 ~]# mongo --host=10.10.10.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://10.10.10.249:28000/
MongoDB server version: 3.6.5
WARNING: shell and server versions do not match
> exit
bye
```

Authentication Restrictions

从 3.6 版本起，新增 authenticationRestrictions 参数用于将数据库用户连接限制为指定的 IP 地址，authenticationRestrictions 参数在下表的创建、更新用户角色中均可使用。

Commands	Methods
createUser	db.createUser()
updateUser	db.updateUser()
createRole	db.createRole()
updateRole	db.updateRole()

例如，我们在服务器中创建一个用户，让只有 192.168.1.251 的客户端能够连接 mongod 服务器 192.168.1.249：

```
> db.createUser(
...     {
...         user: "sys_admin",
...         pwd: "MANAGER",
...         roles: [ { role: "root", db: "admin" } ],
...   authenticationRestrictions: [ {
...         clientSource: ["192.168.1.251"],
...         serverAddress: ["192.168.1.249"]
...       } ]
...     }
... )
```

我们从 HE1（192.168.1.248）访问 mongod 服务器 192.168.1.249 会报认证失败。

```
[root@HE1 ~]# mongo --host=192.168.1.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://192.168.1.249:28000/
MongoDB server version: 3.6.5
WARNING: shell and server versions do not match
> db.auth('sys_admin','MANAGER')
Error: Authentication failed.
0
>
```

而从开启了白名单的 HE4（192.168.1.251）访问 mongod 服务器 192.168.1.249 则正常，这与我们的预期相符。

```
[root@HE4 ~]# mongo --host=192.168.1.249 --port=28000
MongoDB shell version v3.4.14
connecting to: mongodb://192.168.1.249:28000/
MongoDB server version: 3.6.5
WARNING: shell and server versions do not match
> use admin
switched to db admin
> db.auth('sys_admin','MANAGER')
1
```

Change Streams

MongoDB 3.6 支持使用副本集或分片使用 Change Streams，必须是复制协议版本 1，且 WT 存储引擎。

Change Streams 允许应用程序实时了解数据的更改，而不用再去“tail oplog”，应用程序可以使用 Change Streams 来订阅集合上的所有数据更改，并立即响应这些更改。我们可以非常方便地实现一个发布订阅模式。

使用 Change Streams 必须开启 3.6 版本特性参数 featureCompatibilityVersion。

Causal Consistency

在 MongoDB 3.6 以前，如果我们先向 Primary 节点插入一条数据，然后立即向一个 Secondaries 节点查询该条数据，则很可能操作会失败，因为 Primary 节点的数据变化可能尚未同步至 Secondaries 节点。在 3.6 版本中，提出了 Session 的概念，客户端可以预先创建一个 Session，在该 Session 中执行的所有操作会按照顺序执行。

需要客户端使用 MongoDB driver 3.6，以及需要数据库开启 3.6 版本特性参数 featureCompatibilityVersion。

Retryable Writes

如果遇到网络错误，或者在副本集或分片集群中找不到健康的主节点，则可重试写操作，允许 MongoDB 驱动程序重试写操作。重试只会重试 1 次，如果驱动程序无法在目标副本集或分片集群中找到健康的主节点，则驱动程序将等待 serverSelectionTimeoutMS 时间，以便在重试之前确定新的主节点。可重试写入处理故障转移时间不会超过 serverSelectionTimeoutMS（默认为 30s）。

限制：

（1）只有副本集和 Shard 可用。

（2）数据库要求 WT 或 in-memory 存储引擎。

（3）需要客户端使用 MongoDB driver 3.6，以及需要开启 3.6 版本特性参数 featureCompatibilityVersion。

（4）writeconcern 必须配置，i.e{w:0}不可用。

（5）由于重试尝试只进行一次，重试功能可以帮助解决暂时的网络错误，但不能解决持久的网络错误。

（6）驱动程序将等待 serverSelectionTimeoutMS 时间，以在重试之前确定新的主节点。可重试功能不会处理故障转移期超过 serverSelectionTimeoutMS 时间的情况。

> **注意：**如果客户端应用程序在发出写入操作后暂时无法响应 localLogicalSessionTimeoutMinutes，则当客户端应用程序开始响应（不重新启动）时，写入操作可能会重试并重新应用。

serverStatus

serverStatus 新增 logicalSessionRecordCache 项：

```
"logicalSessionRecordCache" : {
        "activeSessionsCount" : 0,
        "sessionsCollectionJobCount" : 5,
        "lastSessionsCollectionJobDurationMillis" : 0,
        "lastSessionsCollectionJobTimestamp" : ISODate("2018-05-31T03:21:34.777Z"),
        "lastSessionsCollectionJobEntriesRefreshed" : 0,
        "lastSessionsCollectionJobEntriesEnded" : 0,
        "lastSessionsCollectionJobCursorsClosed" : 0,
        "transactionReaperJobCount" : 0,
        "lastTransactionReaperJobDurationMillis" : 0,
        "lastTransactionReaperJobTimestamp" : ISODate("2018-05-31T02:56:34.731Z"),
        "lastTransactionReaperJobEntriesCleanedUp" : 0
},
```

2.1 复制集（Replica Sets）

弃用副本集协议版本 0（pv0）。

添加了 replSetResizeOplog 命令来动态调整副本集成员的 oplog 的大小，这适用于运行 WiredTiger 存储引擎的实例。例如：

```
use local
db.oplog.rs.stats().maxSize
use admin
db.adminCommand({replSetResizeOplog:1, size: 16384})
use local
db.oplog.rs.stats().maxSize
```

添加了 catchUpTakeoverDelayMillis 配置选项，指定节点在发起选举之前等待的时间，默认为 30 秒。

对于使用协议版本 1（pv1）的副本集，如果“仲裁人”发现与“候选人”有相同或更高优先级的“人”，则它们将在选举中投反对票。

通过添加 oplogInitialFindMaxSeconds 参数来调整副本集的成员在数据同步期间其 find 命令等待多久，默认为 60 秒。

增加了 waitForSecondaryBeforeNoopWriteMS 参数，用以指定如果 afterClusterTime 大于 oplog 的最近应用时间，则 Secondary 服务器必须等待多长时间，默认为 10 毫秒。

2.2　分片集群（Sharded Clusters）

为 mongos 添加了 ShardingTaskExecutorPoolMaxConnecting 参数，以控制 mongos 将连接添加到 mongod 实例的速率。默认是 2，仅对 mongos 有效。

添加了 orphanCleanupDelaySecs，它确定从源分片中删除迁移块之前的最小延迟。

现在可以对 config 数据库中的 config.system.sessions 集合进行分片。

listdatabase

db.adminCommand({ listDatabases: 1, nameOnly: true})：添加了 nameOnly 执行命令时不会加锁，而不添加则会请求库级锁。

db.adminCommand({ listDatabases: 1, filter: { "name": /^rep/ } }) filter：会过滤想看的数据库，支持正则表达式。

修改了 validate 命令和 db.collection.validate()方法的行为，只有 WiredTiger 存储引擎强制执行检查点，将所有内存中的数据刷新到磁盘，然后验证磁盘上的数据。

dropDatabase 命令会等待“drop”完所有集合的命令并传播到大部分副本集成员后执行。

对于在副本集和分片集群上运行的命令，相应文档包括 operationTime 和$ clusterTime。

Read Concern

新增 available，对于非分片集群，local 和 available 行为是相同的。对于分片群集，available 提供了对分区的更大容忍度，如果分片正在进行块迁移，则可能会返回孤儿文档。

经历了前两章的热身，我们进入第 3 章的学习。第 3 章详细剖析生产中遇到的 MongoDB 棘手故障案例，如果遇到相似案例，则可以很轻松地解决问题。实战经验是需要积累的，我们要学会在工作中养成做记录的好习惯，把一些棘手案例记下来，或者写成博客。既可以把知识分享出去，又可以避免今后再遇到此类问题时束手无策。

第3章 运维实战：故障案例分析

3.1 调整 oplog 大小引发的从库宕机

背景：

我们都知道 oplog 是可以手动调节大小的。由于业务大量写入，导致 oplog window 窗口时间不断降低，过低的窗口时间会给数据库安全性带来隐患，因此我们决定调大 oplog 的值。调大 oplog 的值后，引起了从库宕机。

日志抓取：

```
    2017-08-30T08:32:40.761+0800 I REPL      [replication-2] could not find
member to sync from
    2017-08-30T08:32:40.761+0800 E REPL      [rsBackgroundSync] too stale to
catch up -- entering maintenance mode
    2017-08-30T08:32:40.762+0800 I REPL      [rsBackgroundSync] Our newest
OpTime : { ts: Timestamp 1503977172000|27, t: 1 }
    2017-08-30T08:32:40.762+0800 I REPL     [rsBackgroundSync] Earliest OpTime
```

```
available is { ts: Timestamp 1503998451000|1, t: -1 }
    2017-08-30T08:32:40.762+0800  I  REPL          [rsBackgroundSync]  See
http://dochub.mongodb.org/core/resyncingaverystalereplicasetmember
    2017-08-30T08:32:40.762+0800 I REPL      [rsBackgroundSync] going into
maintenance mode with 1820 other maintenance mode tasks in progress
    2017-08-30T08:33:19.769+0800 I REPL      [rsBackgroundSync] sync source
candidate: 192.168.1.250:28000
    2017-08-30T08:33:19.769+0800 I REPL    [replication-2] We are too stale to
use 192.168.1.250:28000 as a sync source. Blacklisting this sync source because
our last fetched timestamp: 59a4ded4:1b is before their earliest timestamp:
59a524a9:21e for 1min until: 2017-08-30T08:34:19.769+0800
```

原因：

在本案例中存在延迟从库的问题，延迟从库由于延迟，还没有应用到 oplog 就被“drop”了，进而导致延迟从库处于 recovering 状态。

解决方案：

（1）先取消延迟配置，扩容延时从库的 oplog 大小，再扩容主库的 oplog。

（2）对于主库需要先降级再进行升级操作。

oplog 扩容流程：

（1）启动一个新端口，例如，28480，先使用 netstat -anlp|grep 28480 检查是否已存在端口。

（2）将待扩容 oplog 大小的库端口换成 28480。

（3）关库。

（4）注释认证相关，修改 oplog 大小的参数，例如，将 oplog 扩容至 40GB。

```
    vi /home/work/mongodb/mongo_28000/etc/mongodb.conf。
    oplogSizeMB: 40960 #40GB
```

（5）起库。

```
    numactl --interleave=all /home/work/mongodb/3.4.4/bin/mongod -f /home
/work/mongodb/mongo_28000/etc/mongodb.conf。
```

（6）登库。

```
    mongo    127.0.0.1:28480/admin    -u    sys_admin    -p    yourpassword
```

```
--authenticationDatabase admin。
```

（7）执行 oplog recreate 操作。

```
    >use local
    >db = db.getSiblingDB('local')
    >db.temp.drop()
    >db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural :
-1} ).limit(1).next() )
    >db.temp.find()
    >db = db.getSiblingDB('local')
    >db.oplog.rs.drop()
    >db.runCommand( { create: "oplog.rs", capped: true, size: (80 * 1024 * 1024
* 1024) } )
    >db.oplog.rs.save( db.temp.findOne() )
    >db.oplog.rs.find()
```

（8）关库，修改 mongodb.conf 为原始值，启动。

```
    numactl --interleave=all /home/work/mongodb/3.4.4/bin/mongod -f /home/
work/mongodb/mongo_28000/etc/mongodb.conf。
```

3.2 hotbackup 报错

背景：

Percona MongoDB 支持 Hot Backup，解决了 MongoDB 官方版本只能使用 mongodump 的情况。mongodump 在小库中还好，数据量增大后，备份和恢复都是痛苦的。线上进行热备时，由于历史原因 MongoDB 存在多个版本，在使用 MongoDB 热备份老版本数据库时，出现错误导致热备不能正常进行。

日志抓取：

```
    heleitest:PRIMARY>     db.runCommand({createBackup:     1,     backupDir:
"/home/work/backup"})
    {
      "ok" : 0,
      "errmsg" : "boost::filesystem::copy_file: No such file or directory:
\"/home/work/mongodb/data/db_27020/admin/collection/11-6714811575794894766.w
```

```
t\", \"/home/work/backup/admin/collection/11-6714811575794894766.wt\""
    }
```

原因：

为了便于管理，在生产库中配置了如下两个参数。

```
    directoryPerDB: true
    directoryForIndexes: true
```

这两个参数对于热备兼容性存在一定问题。

解决方案：

（1）使用 Percona MongoDB 3.2.12 之后的版本。

（2）如果一定要保留 directoryPerDB 和 directoryForIndexes，则需要重新进行初始化同步。这个过程是非常耗时的，因此推荐升级。

官方升级记录信息：

```
    #PSMDB-123: Fixed the creation of proper subdirectories inside the backup
destination directory
```

3.3　MongoDB 最大连接数限制

背景：

MongoDB 数据库没有配置连接数，应用异常导致连接激增，致使 MongoDB 数据库服务器的 load 值非常高。当然我们可以通过控制 net.maxIncomingConnections 参数来限制 MongoDB 的连接数大小，如果没有配置连接数，那么 MongoDB 的最大连接数就是无限多了吗？答案当然是否定的。

原因：

通过如下命令能看到 MongoDB 进程的 max open files 大小：

```
    ps -ef|grep mongo
    work     134660      1  6 05:48 ?        00:43:28 /mongod -f mongodb.conf

    cat /proc/134660/limits
    Max open files            150240
```

这个值和我们在库中通过 serverstatus()看到的最大连接数有什么关系呢？

```
sysCoreGray01:SECONDARY> db.serverStatus().connections
{ "current" : 2064, "available" : 118128, "totalCreated" : 924119 }
sysCoreGray01:SECONDARY>
```

这里最大连接数实际上和 max open files 有关：

$$118128+2064=120192=150240\times0.8$$

至于是 0.8 的原因是出于一种保护性的考虑，因为不可能把所有的 open file 句柄都拿来维护连接数，还需要保持对磁盘上文件的访问。我们能够在源码中看到关于连接数的限制：

```
int getMaxConnections() {
#ifdef _WIN32
    return DEFAULT_MAX_CONN;
#else
    struct rlimit limit;
    verify(getrlimit(RLIMIT_NOFILE, &limit) == 0);
    int max = (int)(limit.rlim_cur * .8);
    LOG(1) << "fd limit"
<< " hard:" << limit.rlim_max << " soft:" << limit.rlim_cur << " max conn:
" << max;
    return max;
#endif
}
```

解决办法：

（1）配置 net.maxIncomingConnections 参数，限制 MongoDB 的最大连接数。

（2）联系开发排查应用连接异常增高的原因。

3.4 MongoDB 启动失败

背景：

MongoDB 启动报 Failed to unlink socket file /tmp/mongodb-28000.sock Operation not permitted 的错误，说明启动失败。我们建议使用非 root 用户来启停 MongoDB 数据库。

日志抓取：

启动 mongod 进程会看到类似的报错：

```
    about to fork child process, waiting until server is ready for connectio
ns.
    forked process: 12704
    ERROR: child process failed, exited with error number 14
```

日志中有如下报错：

```
    [root@xxx-mgdb01 ~]#  tail -f /home/work/mongodb_sharding/mongo_28000/
log/mongodb.log
    2018-01-09T16:49:16.166+0800 I CONTROL  [initandlisten]     target_arch:
x86_64
    2018-01-09T16:49:16.166+0800 I CONTROL  [initandlisten] options: { config:
    2018-01-09T16:49:16.166+0800 E NETWORK  [initandlisten] Failed to unlink
socket file /tmp/mongodb-28000.sock Operation not permitted
    2018-01-09T16:49:16.166+0800 I -        [initandlisten] Fatal Assertion
28578 at src/mongo/util/net/listen.cpp 195
    2018-01-09T16:49:16.166+0800 I -        [initandlisten]
```

```
2018-01-09T16:49:16.166+0800 W -        [initandlisten] Detected unclean shutdown - /home/work/mongodb_sharding/mongo_
28000/data/mongod.lock is not empty.
2018-01-09T16:49:16.166+0800 E NETWORK  [initandlisten] Failed to unlink socket file /tmp/mongodb-28000.sock Operation
 not permitted
2018-01-09T16:49:16.166+0800 I -        [initandlisten] Fatal Assertion 28578 at src/mongo/util/net/listen.cpp 195
2018-01-09T16:49:16.166+0800 I -        [initandlisten]
```

原因：

如果一直使用非 root 用户启动数据库，但有一次启动时忘记了，使用了 root 用户，那么下次在使用非 root 用户启动数据库时会出现数据库不能启动的情况，这是由于.sock 文件在使用 root 启动后，其属主是 root，而非 root 启动后发现该文件存在却无权操作该文件，所以引起启动失败。

目录文件属主可看到诸如下面的内容：

```
    [root@xxx-mgdb01 tmp]# ll
    total 8
    srwx------ 1 root root    0  Jan  9 16:33  mongodb-28000.sock
    srwx------ 1 work work    0  Dec 15 09:54  mongodb-30000.sock
    srwx------ 1 work work    0  Dec 15 09:53  mongodb-40000.sock
    -rw-r--r-- 1 root root 2083 Jan  9 12:45  tmp_get_port
    drwxr-xr-x 3 root root 4096 Dec 1  18:51  xbox_rename
```

解决办法：

删除.sock 文件，重新使用专用的 MongoDB 账户启动即可。

```
    [root@xxx-mgdb01 tmp]# rm -rf mongodb-28000.sock
    [root@xxx-mgdb01 tmp]# su - work
    [work@xxx-mgdb01 ~]$ numactl --interleave=all /home/work/mongodb_
sharding/3.4.4/bin/mongod -f
/home/work/mongodb_sharding/mongo_28000/etc/mongodb.conf
    about to fork child process, waiting until server is ready for connections.
    forked process: 24331
    child process started successfully, parent exiting
```

3.5 Mongos 异常宕机

背景：

线上 MongoDB 分片集群中路由节点的 mongos 进程某一时刻异常退出。

日志抓取：

```
    2017-12-14T18:01:43.047+0800 I ASIO [NetworkInterfaceASIO-
TaskExecutorPool-19-0] Successfully connected to 192.168.1.250:27017, took
291ms (10 connections now open to 192.168.1.250:27017)
    2017-12-14T18:01:43.048+0800 I ASIO [NetworkInterfaceASIO-
TaskExecutorPool-19-0] Connecting to 192.168.1.252:27017
    2017-12-14T18:01:43.048+0800 I ASIO [NetworkInterfaceASIO-
TaskExecutorPool-19-0] Connecting to 192.168.1.252:27017
    2017-12-14T18:01:43.050+0800 F - [NetworkInterfaceASIO-
TaskExecutorPool-22-0] Failed to mlock: Cannot allocate memory
    2017-12-14T18:01:43.050+0800 I - [NetworkInterfaceASIO-
TaskExecutorPool-22-0] Fatal Assertion 28832 at src/mongo/base/secure_
allocator.cpp 246
    2017-12-14T18:01:43.050+0800 I - [NetworkInterfaceASIO -
TaskExecutorPool -22-0] …
    2017-12-14T18:01:43.054+0800 F - [NetworkInterfaceASIO-
TaskExecutorPool-22-0] Got signal: 6 (Aborted).
```

原因：

大量慢查询导致 mongos 与 mongod 之间的连接长时间没有释放，使得连接池中的连接无法复用，开始新建连接，而新建连接需要分配一定的锁定内存。mongos 需要分配的锁定内存超过 max locked memory 的限制，于是程序崩溃退出。max locked memory 的默认值是 64KB。

解决办法：

（1）升级至 3.4.6 版本以上，这是官方的一个 bug，可以通过升级进行修复。

（2）使用 uimit –l 临时调大 max locked memory 的值来绕开这个问题。

3.6　sharding 集群执行 sh.stopBalancer()命令卡住

背景：

开启 balancer 后，客户反馈前端应用写入缓慢，查询超时。因此我们尝试关闭 balancer，以避免 chunk 迁移对集群性能带来的影响。

但是在调用 sh.stopBalancer 的时候，却发现停不下来，sh.stopBalancer 会处于卡住的状态。

日志抓取：

```
mongos>sh.stopBalancer()
Waiting for active hosts...
Waiting for the balancer lock...
assert.soon failed,msg:Waited too long for lock balancer to unlock
doassert@src/mongo/shell/assert.js:18:14
assert.soon@src/mongo/shell/assert.js:202:13
sh.waitForDLock@src/mongo/shell/utils_sh.js:198:1
sh.waitForBalancerOff@src/mongo/shell/utils_sh.js:264:9
sh.waitForBalancer@src/mongo/shell/utils_sh.js:294:9
sh.stopBalancer@src/mongo/shell/utils_sh.js:161:5
@(shell):1:1
Balancer still may be active, you must manually verify this is not the c
ase using the
config.changelog collection.
2018-02-11T16:28:29.753+0800
E QUERY    [thread1] Error: Error:
assert.soon failed, msg:Waited too long for lock balancer to unlock :
sh.waitForBalancerOff@src/mongo/shell/utils_sh.js:268:15
sh.waitForBalancer@src/mongo/shell/utils_sh.js:294:9
```

```
    sh.stopBalancer@src/mongo/shell/utils_sh.js:161:5
    @(shell):1:1
```

排查：

调用 sh.status()命令能够看到当前 balancer 已经关闭了，但是 running 的状态还是 yes，说明有迁移正在运行。

```
    balancer:
    Currently enabled:  no
    Currently running:  yes
```

通过查看发现 migrations 集合下为空，说明没有集合在迁移：

```
    mongos> db.migrations.find()
```

查看 locks 集合下的信息，处于 2 状态的说明正持有锁：

```
    mongos> db.locks.find()
    { "_id" : "balancer", "state" : 2, "ts" : ObjectId("5a324c42329457086086
da07"), "who" : "ConfigServer:Balancer", "process" : "ConfigServer", "when"
: ISODate("2018-01-31T08:33:43.346Z"), "why" : "CSRS Balancer" }
```

在 locks 文档中只有_id 是 balancer 的处于 state 2，why 阐述了原因是 CSRS Balancer。

locks 集合中的 why 告诉我们持有锁的原因，如果有正在迁移的文档，则其状态应该是 2，why 中的原因会显示 Migrating chunk(s) in collection db.collationname。

从 3.4 版本起，balancer 的状态字段值将始终为 2，以防止老版本的 mongos 实例执行平衡操作。when 字段指 config server 成员成为主节点的时间。

原因：sh.stopBalancer 停不下来，常见的原因有以下几个。

（1）正在做 chunk 迁移，必须等待 chunk 迁移完成后才能正常停止。

（2）后端的 server 时间不同步。

（3）Mongo 客户端版本低于 server 端，本节就是第 3 种情况。Mongo 客户端的版本是 3.2 版本，config server 和 mongod 都是 3.4 版本的 Mongo。由于客户端 Mongo 是 3.2 版本，而 config 节点是 3.4 版本，3.2 版本的 mongos 在执行 stopBalancer()时，如果 balancerStop 命令没有找到，那么它会使用旧版本的逻辑，等待锁被释放。从 3.4 版本起，balance 进程从 mongos 移动至 configer server 的 Primary 节点上。

解决办法：

替换老版本的 mongo 客户端，使用 3.4 版本的客户端：

```
mongos> sh.stopBalancer()
{ "ok" : 1 }
```

可以看到，在使用 3.4 版本的 mongos 后可以正常停止 Balancer 了。

3.7　Remove shard 失败

背景：

在 sharding 中，有一个 shard 机器存在异常，因此希望删掉这个 shard，但执行 remove shard 命令时能看到状态一直是 draining ongoing，这个状态已经持续几天了。

日志抓取：

```
    Cannot move chunk: the maximum number of documents for a chunk is 250001,
the maximum chunk size is 62914560, average document size is 92. Found 368389
documents in chunk  ns: mkj.test1 { num: -8676144648030066959 } -> { num:
-8225288963219700419 }
```

意思是这个 chunk 超过 chunksize 的最大限制 25 万（字节）了，这个 chunk 有 36 万多（字节）的文档。

后面日志能看到：Marking chunk shard: rs1……as jumbo，将这个集合标记为了 jumbo。

```
    2018-01-14T01: 22: 57.089+0800 I SHARDING [Balancer] performing a split
because migration mkj testl: [i num: -8676144648030066959 ,i num:
-82252889632197004
    91),from rsl, to rs2 failed for size reasons : caused by ChunkTooBig: Cannot
move chunk: the maximum number of documents for a chunk is 250001, the max
    imum chunk size is 62914560, average document size is 92. Found 368389
documents in chunk ns: mkj. testl i num: -8676144648030066959->i num:
-82252889632
    19700419}
    2018-01-14T01:22:57.089+0800 I SHARDING [Balancer] Refreshing chunks for
collection mkj testl based on version 1954 15a44e966f11de8caele2dac9
    2018-01-14T01:22:57.090+0800 I SHARDING [CatalogCacheLoader-30] Refresh for
```

```
collection mkj testl took 0 ms and found version 1954 15a44e966fllde8caele2dac9
    2018-01-14T01:22:57.320+0800 I SHARDING [Balancer] Split chunk i splitChunk:
mkj testl", configdb: configSvr/190.168.1.253:30000, 190.168.1.254:30001,
190.168.1.255:30002, from: rsl",key Pattern: i num: hashed"), shardVersion:
Timestamp 19540001, ObjectId( 5a44e966fllde8caele2dac9)
    min:{num:-8676144648030066959],max:i num: -8225288963219700419, splitKeys:
[i num: -8370002822760159629 3I failed : caused by : LockBusy: timed out waiting
for mkj.testl
    2018-01-14T01: 22: 57. 321+0800 I SHARDING [Balancer] Marking chunk shard:
rsl, lastmod:
1954|1|5a44e966f1lde8caele2dac9,({ num:-8676144648030066959},{num:-822514464
8030064359}) as jumbo
```

再次执行 removeshard 命令会一直提示 ongoing，但剩余的几个 chunk 根本就没有进行迁移。

```
mongos> db.adminCommand( { removeShard: "rs1" } )
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
                "chunks" : NumberLong(6),
                "dbs" : NumberLong(1)
        },
        "note" : "you need to drop or movePrimary these databases",
        "dbsToMove" : [
                "helei"
        ],
        "ok" : 1
}
```

使用 sh.status(true)能看到 jumbo chunk，jumbo chunk 是无法进行迁移的：

```
    mongos>sh.status(true)
    mkj.test1
    shard key: { "num" : "hashed" }
    unique: false
    balancing: true
    chunks:
    rs1         19
    rs2         1975
    { "num" : { "$minKey" : 1 } } -->> { "num" : NumberLong("-8916010317837577954") }
on : rs2 Timestamp(987, 0)
    { "num" : NumberLong("-8916010317837577954") } -->> { "num" :
NumberLong("-8915946367386870651") } on : rs2 Timestamp(988, 0)
    { "num" : NumberLong("-8915946367386870651") } -->> { "num" :
```

```
NumberLong("-8915901093971537175") } on : rs2 Timestamp(989, 0)
    .........
    { "num" : NumberLong("-8677007382576487465") } -->> { "num" :
NumberLong("-8676571893798487824") } on : rs2 Timestamp(1953, 0)
    { "num" : NumberLong("-8676571893798487824") } -->> { "num" :
NumberLong("-8676144648030066959") } on : rs2 Timestamp(1954, 0)
    { "num" : NumberLong("-8676144648030066959") } -->> { "num" :
NumberLong("-8225288963219700419") } on : rs1 Timestamp(1964, 1) jumbo
    { "num" : NumberLong("-8225288963219700419") } -->> { "num" :
NumberLong("-7787526991126184118") } on : rs2 Timestamp(218, 2)
    { "num" : NumberLong("-7787526991126184118") } -->> { "num" :
NumberLong("-7351520903994487172") } on : rs2 Timestamp(218, 3)
    { "num" : NumberLong("-7351520903994487172") } -->> { "num" :
NumberLong("-7229595260522338495") } on : rs2 Timestamp(218, 4)
    { "num" : NumberLong("-7229595260522338495") } -->> { "num" :
NumberLong("-7000271679142573352") } on : rs2 Timestamp(4, 4)
    { "num" : NumberLong("-7000271679142573352") } -->> { "num" :
NumberLong("-6453154770799916614") } on : rs1 Timestamp(5, 26) jumbo
    { "num" : NumberLong("-6453154770799916614") } -->> { "num" :
NumberLong("-5905151665570058669") } on : rs1 Timestamp(5, 27) jumbo
    { "num" : NumberLong("-5905151665570058669") } -->> { "num" :
NumberLong("-5895474450038443511") } on : rs2 Timestamp(1955, 0)
    { "num" : NumberLong("-5895474450038443511") } -->> { "num" :
NumberLong("-5345899404183394423") } on : rs1 Timestamp(5, 23) jumbo
    { "num" : NumberLong("-5345899404183394423") } -->> { "num" :
NumberLong("-4798135443173280736") } on : rs1 Timestamp(5, 24) jumbo
    { "num" : NumberLong("-4798135443173280736") } -->> { "num" :
NumberLong("-4788518681703360331") } on : rs2 Timestamp(1956, 0)
    { "num" : NumberLong("-4788518681703360331") } -->> { "num" :
NumberLong("-4761467319058623124") } on : rs2 Timestamp(1957, 0)
    { "num" : NumberLong("-4761467319058623124") } -->> { "num" :
NumberLong("-4611686018427387902") } on : rs2 Timestamp(1958, 0)
    { "num" : NumberLong("-4611686018427387902") } -->> { "num" :
NumberLong("-4265195380846682465") } on : rs1 Timestamp(218, 1994) jumbo
    { "num" : NumberLong("-4265195380846682465") } -->> { "num" :
NumberLong("-3920668983291466572") } on : rs1 Timestamp(218, 1995) jumbo
    { "num" : NumberLong("-3920668983291466572") } -->> { "num" :
NumberLong("-3760130297895645202") } on : rs2 Timestamp(1959, 0)
```

```
    { "num" : NumberLong("-3760130297895645202") } -->> { "num" :
NumberLong("-3414540185060770553") } on : rs1 Timestamp(218, 1989) jumbo
    { "num" : NumberLong("-3414540185060770553") } -->> { "num" :
NumberLong("-3067909645452862499") } on : rs1 Timestamp(218, 1990) jumbo
    { "num" : NumberLong("-3067909645452862499") } -->> { "num" :
NumberLong("-2907288961705410456") } on : rs2 Timestamp(1960, 0)
    { "num" : NumberLong("-2907288961705410456") } -->> { "num" :
NumberLong("-2490792671212143789") } on : rs1 Timestamp(5, 19) jumbo
    { "num" : NumberLong("-2490792671212143789") } -->> { "num" :
NumberLong("-2054870083334942950") } on : rs1 Timestamp(218, 14) jumbo
    { "num" : NumberLong("-2054870083334942950") } -->> { "num" :
NumberLong("-1618154385089705822") } on : rs1 Timestamp(218, 15) jumbo
    { "num" : NumberLong("-1618154385089705822") } -->> { "num" :
NumberLong("-1576783012578929783") } on : rs2 Timestamp(1961, 0)
    { "num" : NumberLong("-1576783012578929783") } -->> { "num" :
NumberLong("-1139213885630926839") } on : rs1 Timestamp(218, 5) jumbo
    { "num" : NumberLong("-1139213885630926839") } -->> { "num" :
NumberLong("-702070166773466028") } on : rs1 Timestamp(218, 6) jumbo
    { "num" : NumberLong("-702070166773466028") } -->> { "num" :
NumberLong("-658393559651725942") } on : rs2 Timestamp(1962, 0)
    { "num" : NumberLong("-658393559651725942") } -->> { "num" :
NumberLong("-3659210059554202l7") } on : rs2 Timestamp(1963, 0)
    { "num" : NumberLong("-3659210059554202l7") } -->> { "num" : NumberLong(0) }
on : rs1 Timestamp(218, 0) jumbo
    { "num" : NumberLong(0) } -->> { "num" : NumberLong("328331916134076295") }
on : rs1 Timestamp(218, 2471) jumbo
    { "num" : NumberLong("328331916134076295") } -->> { "num" :
NumberLong("658668949444102762") } on : rs1 Timestamp(218, 2472) jumbo
    { "num" : NumberLong("658668949444102762") } -->> { "num" :
NumberLong("858639075688516926") } on : rs2 Timestamp(1964, 0)
    { "num" : NumberLong("858639075688516926") } -->> { "num" :
NumberLong("1715911159523493708") } on : rs1 Timestamp(5, 15) jumbo
    { "num" : NumberLong("1715911159523493708") } -->> { "num" :
NumberLong("2138442408633098575") } on : rs1 Timestamp(5, 16) jumbo
    { "num" : NumberLong("2138442408633098575") } -->> { "num" :
NumberLong("2484202611162080916") } on : rs2 Timestamp(218, 1967)
    ......
    { "num" : NumberLong("8567398833422063701") } -->> { "num" :
```

```
NumberLong("8859618968294798195") } on : rs2 Timestamp(5, 10)
    { "num" : NumberLong("8859618968294798195") } -->> { "num" : { "$maxKey" :
1 } } on : rs2 Timestamp(2, 11)
```

解决办法：

把上面的 19 个 jumbo chunk 进行分割：

```
    sh.splitAt( "mkj.test1", { "num": NumberLong("-8476144648030066959") })
```

执行命令比较耗时，所以第一个命令执行后，立即执行后面的命令会有提示：

```
    {
    "code" : 46,
    "ok" : 0,
    "errmsg" : "split failed due to LockBusy: timed out waiting for hl.test1"
    }
    sh.splitAt( "mkj.test1", { "num": NumberLong("-6853154770799916614") })
```

因为执行 split 需要时间，sh.status()也能看到 Currently running:　yes。

查看 locks 集合能够发现正在执行迁移：

```
    mongos> db.locks.find({"state":2})
    {    "_id"    :    "balancer",    "state"    :    2,    "ts"    :    ObjectId
("5a5efcb9045395df3621ced9"), "who" : "ConfigServer:Balancer", "process" :
"ConfigServer", "when" : ISODate("2018-01-17T07:35:37.286Z"), "why" : "CSRS
Balancer" }
    {    "_id"    :    "mkj.test1",    "state"    :    2,    "ts"    :    ObjectId
("5a5efcb9045395df3621ced9"), "who" : "ConfigServer:Balancer", "process" :
"ConfigServer",    "when"    :    ISODate("2018-02-01T06:36:33.188Z"),    "why"    :
"Migrating chunk(s) in collection mkj.test1" }
```

迁移期间能在 config server 的日志中看到获取锁的相关日志。

```
    2018-02-01T15:21:33.777+0800 I SHARPING [Balancer] Refreshing chunks for
collection mkj.test based on version 145|0||5a421fadfIlde8cael05c68f
    2018-02-01T15:21:33.777+0800 I SHARPING [CatalogCacheLoader-42] Refresh for
collection mkj.test took 0 ms and found version 145|0||5a421fadfIlde8cael05c68f
    2018-02-01T15:21:33.777+0800 I SHARPING [Balancer] Refreshing chunks for
```

```
collection mkj.testl based on version 2001|3||5a44e966fIlde8caele2dac9
    2018-02-01T15:21:33.778+0800 I SHARPING [CatalogCacheLoader-42] Refresh for collection mkj.testl took 0 ms and found version 2001|3||5a44e966fIlde8caele2dac$
    2018-02-01T15:21:33.778+0800 I SHARPING [Balancer] Refreshing chunks for collection mkj.test2 based on version 1|0||5a4c8b692e235de285acdcba
    2018-02-01T15:21:33.778+0800 I SHARPING [CatalogCacheLoader-42] Refresh for collection mkj.test2 took 0 ms and found version 1|0||5a4c8b692e235de285acdcba
    2018-02-01T15:21:33.781+0800 I SHARPING [Balancer] Refreshing chunks for collection mkj.testl based on version 2001|3||5a44e966fIlde8caele2dac9
    2018-02-Q1T15:21:33.781+0800 I SHARPING [CatalogCacheLoader-42] Refresh for collection mkj.testl took Q ms and found version 2QQ1|3||5a44e966fIlde8caele2dac^
    Migrating chunk(s) in collection mkj.testl1,
    2018-02-01T15:21:52.057+0800 W NETWORK [ReplicaSetMonitor-TaskExecutor-O] Failed to connect to 10.38.163.111:20000, in(checking socket for error after poll;
    .reason: Connection refused
```

在 sh.status()中能看到 test1 集合在 rs1 中的 chunk 不断减少，这是因为之前开启了 removeShard，它一直处于 draining ongoing 状态，只是由于 jumbo chunk 导致不能被迁移，在将 jumbo chunk 分割后，可以看到 rs1 中的 chunk 数量越来越少，之后就可以移除 rs1。

```
    mkj.test1
    shard key: { "num" : "hashed" }
    unique: false
    balancing: true
    chunks:
    rs1         7
    rs2         1999
    too many chunks to print, use verbose if you want to force print
```

```
mkj.test1
        shard key: { "num" : "hashed" }
        unique: false
        balancing: true
        chunks:
                rs1     7
                rs2     2000
        too many chunks to print, use verbose if you want to force print
```

```
mkj.test1
        shard key: { "num" : "hashed" }
        unique: false
        balancing: true
        chunks:
                rs1     1
                rs2     2012
        too many chunks to print, use verbose if you want to force print
```

```
mkj.test1
        shard key: { "num" : "hashed" }
        unique: false
        balancing: true
        chunks:
                rs2     2014
        too many chunks to print, use verbose if you want to force print
```

3.8　move chunk aborted

背景：

在执行 sh.status()查看集群状态时，我们发现如下情况。

```
    Failed with error 'aborted', from shard01 to shard02
    balancer:
          Currently enabled:  no
          Currently running:  no
              Balancer lock taken at Wed Jan 31 2018 16:33:43 GMT+0800 (CST)
by ConfigServer:Balancer
          Failed balancer rounds in last 5 attempts:  0
          Migration Results for the last 24 hours:
              1 : Failed with error 'aborted', from shard01 to shard 02
```

日志抓取：

```
    2018-02-07T09:20:57.844+0800 W SHARDING [conn10222013] Chunk move failed ::
caused by :: WriteConcernFailed: waiting for replication timed out
    2018-02-07T09:48:28.507+0800 W SHARDING [conn10222013] Chunk move failed ::
caused by :: WriteConcernFailed: waiting for replication timed out
    ......
    2018-02-12T10:35:58.925+0800 W SHARDING [conn10823381] Chunk move failed ::
caused by :: WriteConcernFailed: waiting for replication timed out
```

解决办法如下。

（1）在查看分片状态时能够看到最近的报错。

```
Failed with error 'aborted', from shard01 to shard 02
```

引起这个报错的原因有多种，需要具体问题具体分析，本案例是由于主库大量写入导致从库延迟引起的 aborted。

（2）还有的情况是配置了 wait_for_delete 引起的报错。

```
Chunk move failed :: caused by :: ChunkRangeCleanupPending: can't accept new chunks because there are still 4 deletes from previous migration
```

这是由于同时“movechunk”的数量是有限制的。意思是说，当前正要接收新 chunk 的 shard，并且正在删除上一次迁移出的数据，但是由于不能接收新 chunk，于是本次迁移失败。有时候 shard 的删除持续了十几天都没完成，查看日志可以发现同一个 chunk 的删除在不断重复执行，重启所有无法接受新 chunk 的 shard 可以解决这个问题。

如果采用了 balancer 自动均衡，那么可以加上_waitForDelete 参数，比如：

```
{ "_id" : "balancer", "activeWindow" : { "start" : "12:00", "stop" : "19:30" }, "stopped" : false, "_waitForDelete" : true }
```

这样就不会因 delete 堆积而导致后续 migrate 失败。当然，需要考虑这里的阻塞是否会影响程序正常运转，在实践中要慎重使用 waitForDelete，因为加上它之后迁移性能会非常差，可能出现卡住十几个小时的情况，外界获取了被迁移 chunk 的游标句柄，这时候删除不能执行，阻塞了后续其他迁移操作。

游标被打开而导致被迁移数据无法及时删除时的日志类似于：

```
2015-03-07T10:21:20.118+0800 [RangeDeleter] rangeDeleter waiting for open cursors in: test.test, min: { _id: -6665031702664277348 }, max: { _id: -6651575076051867067 }, elapsedSecs: 6131244, cursors: [ 220477635588 ]
```

这可能会卡住几十个小时，甚至一直卡住，影响后续的 movechunk 操作，导致数据不均衡。

解决方法还是重启。

（3）一个分片只能同时进行 $n/2$ 个 chunk 迁移（n 是 shard 数量）。针对这种情况，可以同时进行 3 个 chunk 迁移。shard 正在接收 chunk，因此不能有新的 movechunk。

```
Chunk move failed :: caused by :: ConflictingOperationInProgress: Unable to
```

```
start new migration because this shard is currently receiving chunk [{ gps_geohash:
"wknyg9tkpbpc" }, { gps_geohash: "wkp0xs67zzzy" }) for namespace
metok_core.geocode_position from syscore_shard2
```

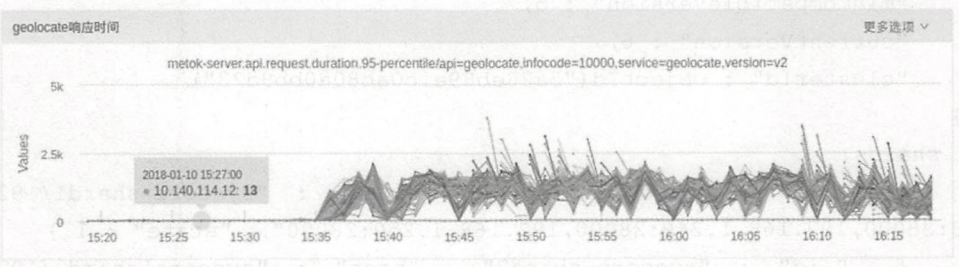
（4）由于 jumbo chunk 导致的。

```
Chunk move failed :: caused by :: ChunkTooBig: Cannot move chunk: the maximum
number of documents for a chunk is 250001, the maximum chunk size is 67108864,
average document size is 324. Found 290203 documents in chunk ns:
metok_core.wifi_position { bssid: -5159092033235030355 } -> { bssid:
-5153591821529598701 }
```

3.9　迁移引发的性能抖动

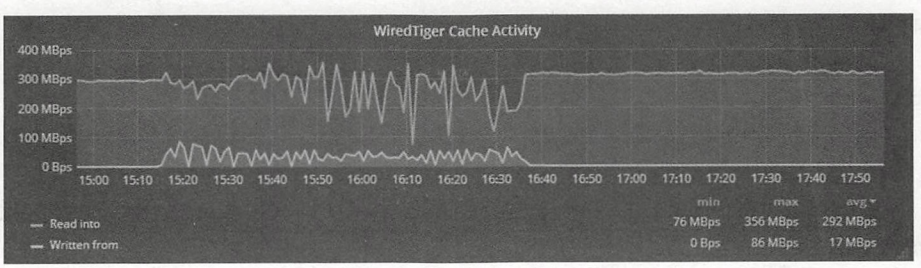
背景：

客户反馈前端应用响应超时，我们观察监控发现存储引擎的 Cache 不断抖动，而且并发很高。日志截图如下图所示。



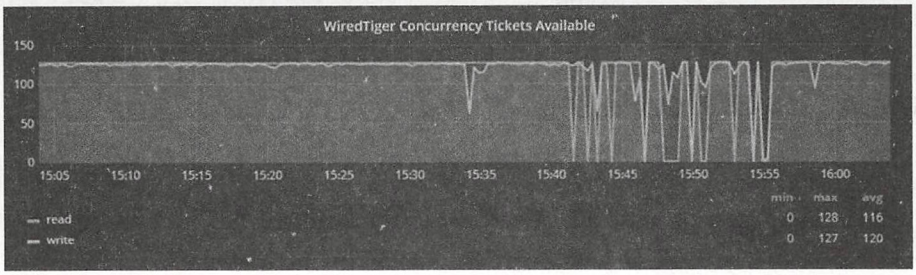
应用反馈从 15:35 起，响应时间明显增长。

监控 MongoDB 的情况发现，磁盘的读出现性能抖动，如下图所示。



下面的线是 write，上面的线是 read。

此时并发非常高，已经把 128 个 ticket 全部取空，如下图所示。



可以看到 ticket 已经取空，此时调用 mongostat，在 ar/aw 列能看到 128，说明有 128 个并发读/写。

在数据库中查看是否有 chunk 迁移，可以看到 Currently running 的状态是 yes：

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("5a28eb89e1c0ab80a0bb9d23")
}
  shards:
    {  "_id" : "syscore_shard1",  "host" : "syscore_shard1/192.168.
1.248:28000,192.168.1.249:28000,192.168.1.250:28000",  "state" : 1 }
    {  "_id" : "syscore_shard2",  "host" : "syscore_shard2/192.168.
1.200:28000,192.168.1.201:28000,190.168.1.202:28000",  "state" : 1 }
    {  "_id" : "syscore_shard3",  "host" : "syscore_shard3/190.168.
1.203:28000,190.168.1.204:28000,190.168.1.205:28000",  "state" : 1 }
  active mongoses:
    "3.4.9-2.9" : 7
  balancer:
    Currently enabled:  yes
    Currently running:  yes
        Balancer lock taken at Fri Dec 15 2017 09:53:35 GMT+0800 (CST) by
ConfigServer:Balancer
    Failed balancer rounds in last 5 attempts:  5
    Last reported error:  could not find host matching read preference { mode:
```

```
"primary" } for set syscore_shard2
        Time of Reported error:  Tue Jan 09 2018 18:57:05 GMT+0800 (CST)
        Migration Results for the last 24 hours:
            No recent migrations
      databases:
        {  "_id" : "metok_core",  "primary" : "syscore_shard3",  "partitioned" :
true }
            metok_core.cell_position
                shard key: { "key" : "hashed" }
                unique: false
                balancing: true
                chunks:
                    syscore_shard1  116
                    syscore_shard2  116
                    syscore_shard3  116
                too many chunks to print, use verbose if you want to force print
            metok_core.geocode_position
                shard key: { "gps_geohash" : 1 }
                unique: false
                balancing: true
                chunks:
                    syscore_shard1  418
                    syscore_shard2  417
                    syscore_shard3  418
                too many chunks to print, use verbose if you want to force print
            metok_core.wifi_position
                shard key: { "bssid" : "hashed" }
                unique: false
                balancing: true
                chunks:
                    syscore_shard1  2780
                    syscore_shard2  2780
                    syscore_shard3  2780
                too many chunks to print, use verbose if you want to force print
        {  "_id" : "test",  "primary" : "syscore_shard2",  "partitioned" :
false }
        {  "_id" : "metok_auth",  "primary" : "syscore_shard1",  "partitioned" :
false }
```

```
mongos>
```

解决方案：

调用 sh.stopBalancer()停止迁移，采取低峰时段进行迁移。我们可以使用如下配置来配置迁移窗口时间（为避免白天操作影响业务，只在凌晨 2 点到 6 点工作）：

```
>use config
>db.settings.update(
>   { _id: "balancer" },
>   { $set: { activeWindow : { start : "02", stop : "06" } } },
>   { upsert: true }
>)
```

3.10 Mongos 连接数异常

背景：

在 sharding 集群中，我们发现 MongoDB 的 ops 异常抖动，但数据库的负载压力并不高，经过检查发现是 mongos 连接数异常引发的后端 ops 异常抖动。

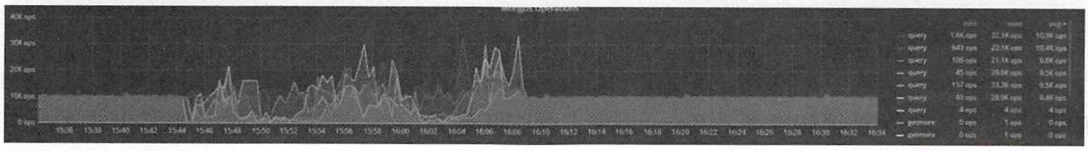
日志抓取：

```
    21935451:2018-01-23T15:44:24.932+0800 I NETWORK  [thread2] connection
accepted from 190.168.1.248:61849 #2152490 (626 connections now open)
    21935452:2018-01-23T15:44:24.932+0800 I NETWORK  [thread2] connection
accepted from 190.168.1.248:61853 #2152491 (627 connections now open)
    21935453:2018-01-23T15:44:24.932+0800 I NETWORK  [thread2] connection
accepted from 190.168.1.248:61857 #2152492 (628 connections now open)
    21935454:2018-01-23T15:44:24.933+0800 I NETWORK  [thread2] connection
accepted from 190.168.1.248:61871 #2152493 (629 connections now open)
```

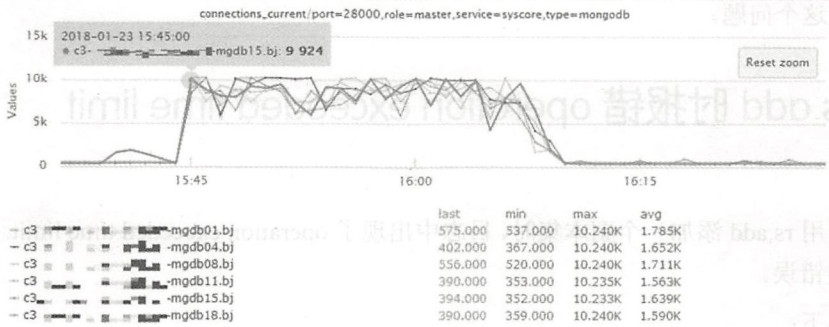
日志中能够看到，越来越多的连接不断被占用：

```
(626 connections now open)
(627 connections now open)
(628 connections now open)
(629 connections now open)
.....
```

通过监控我们发现后端的 mongodbops 出现抖动，如下图所示。



此时我们看一下连接数分布，如下图所示。



通过监控我们发现，在上图 ops 抖动的时间段中，15:44 分时连接数还在 1000 以内的各个主库，15:45 分时连接数达到峰值，正是由于主库连接数接近峰值，所以 mongos 一会能够连接，一会不能够连接，峰值时就会报 HostUnreachable，没到峰值时也会等待当前连接完成后连接 mongos。

我们在日志中能够看到 HostUnreachable 的相关日志：

```
    2018-01-23T15:44:46.062+0800                    I                    ASIO
[NetworkInterfaceASIO-TaskExecutorPool-22-0]          Connecting          to
190.168.1.248:28000
    ......
    2018-01-23T15:44:46.293+0800                    I                    ASIO
[NetworkInterfaceASIO-TaskExecutorPool-7-0] Connecting to 190.168.1.253:28000
    2018-01-23T15:44:46.294+0800                    I                    ASIO
[NetworkInterfaceASIO-TaskExecutorPool-7-0]    Failed    to    connect    to
190.168.1.253:28000 - HostUnreachable: End of file
    2018-01-23T15:44:46.294+0800                    I                    ASIO
[NetworkInterfaceASIO-TaskExecutorPool-7-0] Dropping all pooled connections to
190.168.1.253:28000 due to failed operation on a connection
    2018-01-23T15:44:46.294+0800                    I                    ASIO
[NetworkInterfaceASIO-TaskExecutorPool-7-0] Failed to close stream: Transport
```

```
endpoint is not connected
```

解决办法：

在 sharding 架构中，我们应当使用 net.maxIncomingConnections 参数对 mongos 和 mongod 的最大连接数进行管理和限制，在 mongos 中限制其最大连接，mongod 的连接数应大于所有 mongos 的最大连接数之和。如果 mongos 连接数的上限很大，而 monogd 连接数的上限配置得很小，就会出现这个问题。本案例中，mongod 配置了 10240 连接数上限，而并没有限制 mongos，因此导致了这个问题。

3.11 rs.add 时报错 operation exceeded time limit

背景：

我们使用 rs.add 添加一个副本集时，日志中出现了 operation exceeded time limit. Last fetched optime 这类错误。

日志如下：

```
    2017-09-13T09:27:41.200+0800 I REPL     [replication-120] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1505266015000|4278, t:
4 }[-6153155308264681230]. Restarts remaining: 3
    2017-09-13T09:27:49.866+0800 I REPL     [replication-120] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1505266015000|4378, t:
4 }[4488512953717375549]. Restarts remaining: 3
    2018-01-30T14:16:51.038+0800 I REPL     [replication-6367] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1517292999000|2281, t:
27 }[-9040353305641131992]. Restarts remaining: 3
    2018-01-30T14:16:51.038+0800 I REPL     [replication-6367] Scheduled new
oplog query Fetcher source: 190.168.1.253:28000 database: local query: { find:
"oplog.rs", filter: { ts: { $gte: Timestamp 1517292999000|2281 } }, tailable:
true, oplogReplay: true, awaitData: true, maxTimeMS: 60000, term: 27 } query
metadata: { $replData: 1, $oplogQueryData: 1, $ssm: { $secondaryOk: true } } active:
1 timeout: 65000ms shutting down?: 0 first: 1 firstCommandScheduler:
RemoteCommandRetryScheduler request: RemoteCommand 387246893 --
target:190.168.1.253:28000 db:local cmd:{ find: "oplog.rs", filter: { ts: { $gte:
```

```
Timestamp 1517292999000|2281 } }, tailable: true, oplogReplay: true, awaitData:
true, maxTimeMS: 60000, term: 27 } active: 1 callbackHandle.valid: 1
callbackHandle.cancelled: 0 attempt: 1 retryPolicy: RetryPolicyImpl maxAttempts:
1 maxTimeMillis: -1ms
```

日志中有大量的类似 error: ExceededTimeLimit: operation exceeded time limit. Last fetched optime (with hash): { ts: Timestamp 1505266015000|4278, t: 4 }[-6153155308264681230]. Restarts remaining: 3 这种错误。

解决办法：

如果主库压力较大，从库拉取 oplog 所用时间超过 maxTimeMs，则超时，造成 oplog 传输效率低下，此时应该调大 maxTimeMs 值，从而提高数据拉取效率。升级到 3.4.11 版本可以配置超时时间，或者在低峰期进行 rs.add 操作。

3.12　副本集延迟突然增大到上万秒

背景：

3.2 版本 mongod 副本集架构，一主一从一延迟节点，延迟节点配置了延迟 28800 秒（8 小时），集群业务量很小。

我们收到告警，显示延迟达到了 20000 多秒，而且并不是延迟节点，是普通的 Secondary。

原因：

在 3.2 版本中，如果数据库长时间未写入，那么突然有写入的时候会报延迟瞬间增大，这是由于 oplog 不具备 no-op 导致的。

解决方案：

3.4 新版本中具备了 no-op 操作，可以不断地让各个成员执行 no-op 这个不具备任何实际操作的命令，以避免延迟误报的问题。升级至 3.4 版本可以解决该问题。

3.13　升级发现 infoMessage 异常

背景：

在做完一次升级操作后，我们发现 rs.status()返回的信息中，infoMessage 列出现了 could not find member to sync from 的错误。

日志状态：

```
"members" : [
    {
            "_id" : 0,
            "name" : "192.168.1.248:27020",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 121,
            "optime" : {
            "ts" : Timestamp(1501747251, 50),
            "t" : NumberLong(5)
            },
            "optimeDate" : ISODate("2017-08-03T08:00:51Z"),
            "infoMessage" : "could not find member to sync from",
            "configVersion" : 5,
            "self" : true
    }
```

原因：

这时升级成功了，stateStr 状态也是 Secondary，但 infoMessage 出现异常，这个异常有误导是可以忽略的，出现这个错误的原因是主库没有写入，主从状态是一致的，oplog 不存在最新，因此会有这个提示。一旦主库有写入，这个信息就会消失。预计在 MongoDB 3.5 版本修复。

3.14 对已存在集合 shardcollection 失败

在生产环境中，对一个已有集合开启分片可以使用如下命令，但是却报错：

```
sh.shardCollection("metok_core.wifi_position",{bssid:"hashed"})
"errmsg" : "sharding not enabled for db metok_core"
```

而这个库已经使用如下命令开启分片了：

```
sh.enableSharding("databasename")
sh.shardCollection( "database.collection", { <field> : "hashed" } )
```

原因分析：

这是因为要求整个 shard 对于非空集合必须都具备索引，我们在对已有集合进行分片时，需要先创建索引。主库建完了，从库还没建完，所以导致刚刚的命令执行失败。

这是由于延迟从库导致的，我们可以先取消延迟从库，再执行 sh.shardCollection 命令就可以了：

```
    sh.shardCollection("metok_core.wifi_position",{bssid:"hashed"})
"OK"
```

3.15　operation exceeded time limit

在生产环境中，我们使用 rs.add 添加一个副本集成员时，错误日志中出现了非常多的如下内容：

```
    2017-09-13T09:27:41.200+0800 I REPL     [replication-120] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1505266015000|4278, t:
4 }[-6153155308264681230]. Restarts remaining: 3
    2017-09-13T09:27:49.866+0800 I REPL     [replication-120] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1505266015000|4378, t:
4 }[4488512953717375549]. Restarts remaining: 3

    2018-01-30T14:16:51.038+0800 I REPL     [replication-6367] Restarting oplog
query due to error: ExceededTimeLimit: operation exceeded time limit. Last
fetched optime (with hash): { ts: Timestamp 1517292999000|2281, t:
27 }[-9040353305641131992]. Restarts remaining: 3
    2018-01-30T14:16:51.038+0800 I REPL     [replication-6367] Scheduled new
oplog query Fetcher source: 10.136.30.33:28000 database: local query: { find:
"oplog.rs", filter: { ts: { $gte: Timestamp 1517292999000|2281 } }, tailable:
true, oplogReplay: true, awaitData: true, maxTimeMS: 60000, term: 27 } query
metadata: { $replData: 1, $oplogQueryData: 1, $ssm: { $secondaryOk: true } } active:
1 timeout: 65000ms shutting down?: 0 first: 1 firstCommandScheduler:
RemoteCommandRetryScheduler request: RemoteCommand 387246893 --
target:10.136.30.33:28000 db:local cmd:{ find: "oplog.rs", filter: { ts: { $gte:
Timestamp 1517292999000|2281 } }, tailable: true, oplogReplay: true, awaitData:
```

```
true, maxTimeMS: 60000, term: 27 } active: 1 callbackHandle.valid: 1
callbackHandle.cancelled: 0 attempt: 1 retryPolicy: RetryPolicyImpl maxAttempts:
1 maxTimeMillis: -1ms
```

通过如下图所示的代码逻辑进行处理。

```
if (!responseStatus.isOK()) {
    {
        // We have to call into replication coordinator outside oplog fetcher's mutex.
        // It is OK if the current term becomes stale after this line since requests
        // to remote nodes are asynchronous anyway.
        auto currentTerm =
            _dataReplicatorExternalState->getCurrentTermAndLastCommittedOpTime().value;
        stdx::lock_guard<stdx::mutex> lock(_mutex);
        if (_isShuttingDown_inlock()) {
            log() << "Error returned from oplog query while canceling query: "
                  << redact(responseStatus);
        } else if (_fetcherRestarts == _maxFetcherRestarts) {
            log() << "Error returned from oplog query (no more query restarts left): "
                  << redact(responseStatus);
        } else {
            log() << "Restarting oplog query due to error: " << redact(responseStatus)
                  << ". Last fetched optime (with hash): " << _lastFetched
                  << ". Restarts remaining: " << (_maxFetcherRestarts - _fetcherRestarts);
            _fetcherRestarts++;
            // Destroying current instance in _shuttingDownFetcher will possibly block.
            _shuttingDownFetcher.reset();
            // Move the old fetcher into the shutting down instance.
            _shuttingDownFetcher.swap(_fetcher);
            // Create and start fetcher with current term and new starting optime.
            _fetcher = _makeFetcher(currentTerm, _lastFetched.opTime);
            auto scheduleStatus = _scheduleFetcher_inlock();
            if (scheduleStatus.isOK()) {
                log() << "Scheduled new oplog query " << _fetcher->toString();
                return;
            }
            error() << "Error scheduling new oplog query: " << redact(scheduleStatus)
                    << ". Returning current oplog query error: " << redact(responseStatus);
        }
    }
    _finishCallback(responseStatus);
    return;
} « end if !responseStatus.isOK(... »

    if (hasDeadline() && waitStatus == stdx::cv_status::timeout && deadline == getDeadline()) {
        // It's possible that the system clock used in stdx::condition_variable::wait_until
        // is slightly ahead of the FastClock used in checkForInterrupt. In this case,
        // we treat the operation as though it has exceeded its time limit, just as if the
        // FastClock and system clock had agreed.
        markKilled(ErrorCodes::ExceededTimeLimit);
        return Status(ErrorCodes::ExceededTimeLimit, "operation exceeded time limit");
    }
```

我们可以将问题进行定位并解决。

定位：

如果主库压力较大，从库拉取 oplog 所用时间超过 maxTimeMs，则超时，造成 oplog 传输效率低下，此时应该调大 maxTimeMs 值，提高数据拉取效率。

解决：

升级到 3.4.11 版本可以配置超时时间。

3.16　强制重新配置副本集

背景：

一主两从，其中一从宕机，新的从库加进来，为了替代宕机的从库，但恢复的时候发现是逻辑备份，因此“kill”掉，重新使用物理备份进行恢复。此时，副本集共 4 个成员，2 个宕机，主库降级，剩余的都是 Secondary。

第一想法是删除一个坏掉的从库达到基数，而 rs.remove 要求在 Primary 上执行，此时主库降级，集群中只有两个 Secondary，也就是集群中现在没有任何的 Primary 节点，导致不能够使用 rs.remove 命令，整个集群处于只读状态，不能对外提供写服务。

这时我们应尽快使用强制重新配置命令，来避免这一情况持续过久导致应用异常。

```
cfg = rs.conf()
cfg.members = [cfg.members[0] , cfg.members[2]]
rs.reconfig(cfg, {force : true})
```

强制重配后会导致 version 版本号过高，这个要注意。强制重配后，会出现 version 版本号过高的问题，如果有异常操作导致高版本号（version）的从库被加入低版本号的同副本集中，则会导致集群中的低版本号从库旧节点被“remove”，并变成 other 状态。

```
heleitest:PRIMARY> cfg = rs.conf()
{
        "_id" : "heleitest",
        "version" : 78,
        "protocolVersion" : NumberLong(1),
        "members" : [
            {
                    "_id" : 0,
                    "host" : "192.168.1.248:27017",
                    "arbiterOnly" : false,
                    "buildIndexes" : true,
                    "hidden" : false,
                    "priority" : 2,
                    "tags" : {

                    },
                    "slaveDelay" : NumberLong(0),
```

```
                "votes" : 1
        },
        {
                "_id" : 2,
                "host" : "192.168.1.251:27017",
                "arbiterOnly" : false,
                "buildIndexes" : true,
                "hidden" : false,
                "priority" : 1,
                "tags" : {

                },
                "slaveDelay" : NumberLong(0),
                "votes" : 1
        },
        {
                "_id" : 5,
                "host" : "192.168.1.249:27017",
                "arbiterOnly" : false,
                "buildIndexes" : true,
                "hidden" : false,
                "priority" : 1,
                "tags" : {

                },
                "slaveDelay" : NumberLong(0),
                "votes" : 1
        },
        {
                "_id" : 6,
                "host" : "192.168.1.248:27018",
                "arbiterOnly" : false,
                "buildIndexes" : true,
                "hidden" : false,
                "priority" : 1,
                "tags" : {

                },
```

```
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        },
        {
            "_id" : 7,
            "host" : "192.168.1.249:27018",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {

            },
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        },
        {
            "_id" : 13,
            "host" : "192.168.1.251:27023",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {

            },
            "slaveDelay" : NumberLong(0),
            "votes" : 1
        },
        {
            "_id" : 15,
            "host" : "192.168.1.251:27021",
            "arbiterOnly" : false,
            "buildIndexes" : true,
            "hidden" : false,
            "priority" : 1,
            "tags" : {
```

```
                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
                    }
                ],
                "settings" : {
                        "chainingAllowed" : true,
                        "heartbeatIntervalMillis" : 2000,
                        "heartbeatTimeoutSecs" : 10,
                        "electionTimeoutMillis" : 10000,
                        "catchUpTimeoutMillis" : 2000,
                        "getLastErrorModes" : {

                        },
                        "getLastErrorDefaults" : {
                            "w" : 1,
                            "wtimeout" : 0
                        },
                        "replicaSetId" : ObjectId("5ad5b49994eb742eff1ca45a")
                        }
                }
    heleitest:PRIMARY> cfg.members = [cfg.members[0] , cfg.members
[1],cfg.members[2]]
    [
                {
                        "_id" : 0,
                        "host" : "192.168.1.248:27017",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 2,
                        "tags" : {

                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
                },
                {
```

```
                    "_id" : 2,
                    "host" : "192.168.1.251:27017",
                    "arbiterOnly" : false,
                    "buildIndexes" : true,
                    "hidden" : false,
                    "priority" : 1,
                    "tags" : {

                    },
                    "slaveDelay" : NumberLong(0),
                    "votes" : 1
                },
{
                    "_id" : 5,
                    "host" : "192.168.1.249:27017",
                    "arbiterOnly" : false,
                    "buildIndexes" : true,
                    "hidden" : false,
                    "priority" : 1,
                    "tags" : {

                    },
                    "slaveDelay" : NumberLong(0),
                    "votes" : 1
                }
]
heleitest:PRIMARY> rs.reconfig(cfg, {force : true})
{ "ok" : 1 }
heleitest:PRIMARY> rs.conf()
{
        "_id" : "heleitest",
        "version" : 65271,
        "protocolVersion" : NumberLong(1),
        "members" : [
            {
                    "_id" : 0,
                    "host" : "192.168.1.248:27017",
                    "arbiterOnly" : false,
```

```
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 2,
                        "tags" : {

                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
    },
    {
                        "_id" : 2,
                        "host" : "192.168.1.251:27017",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
},
{
                        "_id" : 5,
                        "host" : "192.168.1.249:27017",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
            }
    ],
    "settings" : {
          "chainingAllowed" : true,
```

```
        "heartbeatIntervalMillis" : 2000,
        "heartbeatTimeoutSecs" : 10,
        "electionTimeoutMillis" : 10000,
        "catchUpTimeoutMillis" : 2000,
        "getLastErrorModes" : {

        },

        "getLastErrorDefaults" : {
              "w" : 1,
              "wtimeout" : 0
          },
          "replicaSetId" : ObjectId("5ad5b49994eb742eff1ca45a")
      }
}
```

3.17　create index oom

一个集合最多只能有 64 个索引，前台创建一个索引单条最大为 100MB 内存，早期调用 createindexes 最高可达到 6.4GB。容易引起 MongoDB 实例 page fault 甚至 OOM。

前台创建索引会阻塞一个库里的所有操作（read+write），但多个库可以同时在前台创建索引。

从 3.4 版本起有了 maxIndexBuildMemoryUsageMegabytes 参数，使用 createindexes，所有索引最大默认为 500MB，可以限制单条索引创建内存的大小。

在 3.4 版本中创建索引时可以看到最大 500MB 的限制日志：

```
    2018-01-22T17:07:23.200+0800    I    COMMAND        [conn120]    command
admin.system.users command: saslStart { saslStart: 1, mechanism: "SCRAM-SHA-1",
payload: "xxx" } numYields:0 reslen:164 locks:{ Global: { acquireCount: { r: 2 },
acquireWaitCount: { r: 1 }, timeAcquiringMicros: { r: 604889 } }, Database:
{ acquireCount: { r: 1 } }, Collection: { acquireCount: { r: 1 } } }
protocol:op_command 606ms
    2018-01-22T17:07:23.201+0800 I INDEX    [repl writer worker 4] build index
on: mi_music.kanjian_song properties: { v: 1, key: { album_id: 1 }, name:
"album_id_1", ns: "mi_music.kanjian_song" }
    2018-01-22T17:07:23.201+0800   I   INDEX          [repl   writer   worker
```

```
4]          building index using bulk method; build may temporarily use up to
500 megabytes of RAM
    2018-01-22T17:07:25.629+0800 I INDEX    [repl writer worker 4] build index
done.  scanned 902308 total records. 2 secs
    2018-01-22T17:07:25.631+0800 I INDEX    [repl writer worker 6] build index
on: mi_music.kanjian_song properties: { v: 1, key: { song_id: 1 }, name:
"song_id_1", ns: "mi_music.kanjian_song" }
    2018-01-22T17:07:25.631+0800   I   INDEX          [repl   writer   worker
6]          building index using bulk method; build may temporarily use up to
500 megabytes of RAM
    2018-01-22T17:07:27.691+0800 I INDEX    [repl writer worker 6] build index
done.  scanned 902308 total records. 2 secs
```

3.18 rs.remove 导致从节点 crash

在副本集架构中，我们经常会通过 rs.add()、rs.remove()命令来调整后台数据库架构。在本案例中，我们异常触发了一个 MongoDB 的 bug，并尽快找到官方的人进行咨询。在生产环境中做实例迁移时，将研发自行维护的 MongoDB 副本集迁移至 DBA 管理，由于硬件和版本都不符合规范，因此我们对集群先进行了升级处理，又使用了 rs.add()和 rs.remove()来完成数据库的迁移工作。

研发自行维护的数据库版本为 2.6 版本，我们先将数据库升级至 3.4 版本，并利用 3.4 版本的特性实现 0 downtime 开启认证。在研发的数据库原有架构中存在离线节点，即 hidden 节点和 no-vote 节点。也正是由于这一特性，触发了 MongoDB 宕机。

原有集群为 7 节点副本集架构，其中 2 台为 hidden 节点，并且配置了 no-vote 参数。

我们利用新的机器使用 rs.add()加入原有副本集，在原有副本集的基础上添加了新的节点，待同步完成后，“rs.remove()”老的研发机器，完成实例迁移。

在使用 rs.remove 时，我们提前写好了迁移文档，由于 rs.remove()的速度很快，采取直接“rs.remove()”多个节点的复制粘贴方式，这也是后续导致 crash 的原因之一。

```
    rs.remove()
    2018-04-17T14:54:23.793+0800  I  NETWORK   [conn163572]  received  client
metadata from 192.168.1.100:16400 conn163572: { driver: { name: "PyMongo",
version: "3.5.1" }, os: { type: "Linux", name: "CentOS 6.4 Final", architecture:
"x86_64", version: "2.6.32-279.23.1.mi5.el6.x86_64" }, platform: "CPython
2.7.6.final.0" }
```

```
    2018-04-17T14:54:23.811+0800 I NETWORK  [thread1] connection accepted from
192.168.1.101:57568 #163573 (73 connections now open)
    2018-04-17T14:54:23.811+0800 I NETWORK  [conn163573] received client
metadata from 192.168.1.101:57568 conn163573: { driver: { name: "PyMongo",
version: "3.5.1" }, os: { type: "Linux", name: "CentOS 6.3 Final", architecture:
"x86_64", version: "2.6.32-279.23.1.mi5.el6.x86_64" }, platform: "CPython
2.7.6.final.0" }
    2018-04-17T14:54:23.818+0800 I -        [replication-25230] Invariant
failure i <_members.size() src/mongo/db/repl/repl_set_config.cpp 620
    2018-04-17T14:54:23.818+0800 I -       [replication-25230]
    ***aborting after invariant() failure

    2018-04-17T14:54:23.822+0800 I NETWORK  [thread1] connection accepted from
192.168.1.102:32210 #163574 (74 connections now open)
    2018-04-17T14:54:23.822+0800 I NETWORK  [conn163574] received client
metadata from 192.168.1.102:32210 conn163574: { driver: { name: "PyMongo",
version: "3.5.1" }, os: { type: "Linux", name: "CentOS 6.3 Final", architecture:
"x86_64", version: "2.6.32-279.23.1.mi5.el6.x86_64" }, platform: "CPython
2.7.6.final.0" }
    2018-04-17T14:54:23.822+0800 I NETWORK  [thread1] connection accepted from
192.168.1.101:57569 #163575 (75 connections now open)
    2018-04-17T14:54:23.823+0800 I NETWORK  [thread1] connection accepted from
192.168.1.103:50661 #163576 (76 connections now open)
    2018-04-17T14:54:23.824+0800 I NETWORK  [conn163576] received client
metadata from 192.168.1.103:50661 conn163576: { driver: { name: "PyMongo",
version: "3.5.1" }, os: { type: "Linux", name: "CentOS 6.3 Final", architecture:
"x86_64", version: "2.6.32-279.23.1.mi5.el6.x86_64" }, platform: "CPython
2.7.6.final.0" }
    2018-04-17T14:54:23.830+0800 I NETWORK  [thread1] connection accepted from
192.168.1.101:57570 #163577 (77 connections now open)
    2018-04-17T14:54:23.830+0800 I NETWORK  [conn163577] received client
metadata from 192.168.1.101:57570 conn163577: { driver: { name: "PyMongo",
version: "3.5.1" }, os: { type: "Linux", name: "CentOS 6.3 Final", architecture:
"x86_64", version: "2.6.32-279.23.1.mi5.el6.x86_64" }, platform: "CPython
2.7.6.final.0" }
    2018-04-17T14:54:23.832+0800 I NETWORK  [thread1] connection accepted from
192.168.1.104:41163 #163578 (78 connections now open)
    2018-04-17T14:54:23.838+0800 F -       [replication-25230] Got signal: 6
```

```
(Aborted).
     0x7ffc3f0a41d1     0x7ffc3f0a3159     0x7ffc3f0a363d     0x7ffc3c0bf500
0x7ffc3bd4f8a5  0x7ffc3bd51085  0x7ffc3e2d7a8e  0x7ffc3ea5883c  0x7ffc3ea58999
0x7ffc3eb19b9d  0x7ffc3eaa2676  0x7ffc3e9c83e9  0x7ffc3e9b0032  0x7ffc3ea41bf7
0x7ffc3e3812f1  0x7ffc3ee25a5a  0x7ffc3ee28e53  0x7ffc3ee2932b  0x7ffc3f0185bc
0x7ffc3f01906c 0x7ffc3f019a56 0x7ffc3fdc7a00 0x7ffc3c0b7851 0x7ffc3be0511d
    ----- BEGIN BACKTRACE -----
    {"backtrace":[{"b":"7FFC3DA22000","o":"16821D1","s":"_ZN5mongo15printSta
ckTraceERSo"},{"b":"7FFC3DA22000","o":"1681159"},{"b":"7FFC3DA22000","o":"16
8163D"},{"b":"7FFC3C0B0000","o":"F500"},{"b":"7FFC3BD1D000","o":"328A5","s":
"gsignal"},{"b":"7FFC3BD1D000","o":"34085",…."190D45F6743DEF9DF8169D65801D45
75B01825BD" }, { "b" : "7FC5BF510000", "path" : "/lib64/libfreebl3.so",
"elfType" : 3, "buildId" : "68195872ECFB188389D29AAF01031A976FD18168" }, { "b" :
"7FC5BEB05000", "path" : "/lib64/libkrb5support.so.0", "elfType" : 3, "buildId" :
"DAE2A7E4E8B37D43EF6839FF5D8E012AFCF21A69" }, { "b" : "7FC5BF902000", "path" :
"/lib64/libkeyutils.so.1",      "elfType"      :      3,      "buildId"      :
"8A8734DC37305D8CC2EF8F8C3E5EA03171DB07EC" }, { "b" : "7FC5C16E3000", "path" :
"/lib64/libselinux.so.1",      "elfType"      :      3,      "buildId"      :
"A287DC6B86A9823038F057105CE64671E0B392EC" } ] }}
     mongod(_ZN5mongo15printStackTraceERSo+0x41) [0x7ffc3f0a41d1]
     mongod(+0x1681159) [0x7ffc3f0a3159]
     mongod(+0x168163D) [0x7ffc3f0a363d]
     libpthread.so.0(+0xF500) [0x7ffc3c0bf500]
     libc.so.6(gsignal+0x35) [0x7ffc3bd4f8a5]
     libc.so.6(abort+0x175) [0x7ffc3bd51085]
     mongod(_ZN5mongo17invariantOKFailedEPKcRKNS_6StatusES1_j+0x0)
[0x7ffc3e2d7a8e]
     mongod(+0x103683C) [0x7ffc3ea5883c]
     mongod(+0x1036999) [0x7ffc3ea58999]

mongod(_ZNK5mongo4repl23TopologyCoordinatorImpl22shouldChangeSyncSourceERKNS
_11HostAndPortERKNS0_6OpTimeERKNS_3rpc15ReplSetMetadataEN5boost8optionalINS8
_18OplogQueryMetadataEEENS_6Date_tE+0x32D) [0x7ffc3eb19b9d]

mongod(_ZN5mongo4repl26ReplicationCoordinatorImpl22shouldChangeSyncSourceERK
NS_11HostAndPortERKNS_3rpc15ReplSetMetadataEN5boost8optionalINS5_18OplogQuer
yMetadataEEE+0xB6) [0x7ffc3eaa2676]
```

```
mongod(_ZN5mongo4repl31DataReplicatorExternalStateImpl18shouldStopFetchingER
KNS_11HostAndPortERKNS_3rpc15ReplSetMetadataEN5boost8optionalINS5_18OplogQue
ryMetadataEEE+0x59) [0x7ffc3e9c83e9]
     mongod(+0xF8E032) [0x7ffc3e9b0032]

mongod(_ZN5mongo4repl12OplogFetcher9_callbackERKNS_10StatusWithINS_7Fetcher1
3QueryResponseEEEPNS_14BSONObjBuilderE+0x1E57) [0x7ffc3ea41bf7]
    mongod(_ZN5mongo7Fetcher9_callbackERKNS_8executor12TaskExecutor25RemoteC
ommandCallbackArgsEPKc+0x621) [0x7ffc3e3812f1]
     mongod(+0x1403A5A) [0x7ffc3ee25a5a]

mongod(_ZN5mongo8executor22ThreadPoolTaskExecutor11runCallbackESt10shared_pt
rINS1_13CallbackStateEE+0x1B3) [0x7ffc3ee28e53]
     mongod(+0x140732B) [0x7ffc3ee2932b]

mongod(_ZN5mongo10ThreadPool10_doOneTaskEPSt11unique_lockISt5mutexE+0x14C)
[0x7ffc3f0185bc]
     mongod(_ZN5mongo10ThreadPool13_consumeTasksEv+0xBC) [0x7ffc3f01906c]

mongod(_ZN5mongo10ThreadPool17_workerThreadBodyEPS0_RKNSt7__cxx1112basic_str
ingIcSt11char_traitsIcESaIcEEE+0x96) [0x7ffc3f019a56]
     mongod(+0x23A5A00) [0x7ffc3fdc7a00]
     libpthread.so.0(+0x7851) [0x7ffc3c0b7851]
     libc.so.6(clone+0x6D) [0x7ffc3be0511d]
    -----  END BACKTRACE  -----
```

可以看到，在错误日志中存在如下内容：

```
    Invariant failure i < _members.size() src/mongo/db/repl/repl_set_config.cpp
620
```

通过追寻源码，我们在 repl_set_config.cpp 的 620 行定位到该处逻辑，内容如下：

```
const MemberConfig& ReplSetConfig::getMemberAt(size_t i) const {
    invariant(i < _members.size());
    return _members[i];
}
```

透过 invariant 关键字，我们向上追寻，可以找到如下内容。

```
#define MONGO_invariant(_Expression)                                    \
    do {                                                                \
        if (MONGO_unlikely(!(_Expression))) {                           \
            ::mongo::invariantFailed(#_Expression, __FILE__, __LINE__); \
        }                                                               \
    } while (false)

#define invariant MONGO_invariant

// Behaves like invariant in debug builds and is compiled out in release. Use for checks, which can
// potentially be slow or on a critical path.

NOINLINE_DECL void invariantFailed(const char* expr, const char* file, unsigned line) noexcept {
    log() << "Invariant failure " << expr << ' ' << file << ' ' << dec << line << endl;
    breakpoint();
    log() << "\n\n***aborting after invariant() failure\n\n" << endl;
    std::abort();
}
```

看完代码逻辑，我们认为报错是由于批量执行 rs.remove()引起的，代码逻辑中可以看出有一个判断：

```
const MemberConfig& ReplSetConfig::getMemberAt(size_t i) const

{ invariant(i < _members.size()); return _members[i]; }
```

例如，一个 5 节点的副本集，其_ID 索引值为 0、1、2、3、4，此时_members.size 是 5， 4 must <5 的逻辑是正确的。

但当我们使用 rs.remove()命令删除超过 1 个节点的成员时，例如，同时删除_id=3 和_id=4，此时副本集就只有 3 个成员，_members.size 变为 3。

但是 i 值却依旧因为某些原因停留在 4，4<3 这个逻辑不对，导致被删除节点 abort 退出 crash。

关于 rs.remove no-vote 节点官方已经定义为 bug，并计划在 4.1 版本中修复。

本章介绍遇到 MongoDB 数据库性能瓶颈的时候，我们该如何解决相关的问题。在大数据时代，大数据量的处理已经成为考量一个数据库的重要标准之一。而 MongoDB 的一个主要目标就是尽可能地让数据库保持卓越的性能，这在很大程度上决定了 MongoDB 的设计。在一个以传统机械硬盘为主导的年代，硬盘很可能会成为性能的短板，而 MongoDB 选择了最大程度利用内存资源作缓存来换取卓越的性能，并且会自动选择速度最快的索引来进行查询。MongoDB 尽可能精简数据库，将尽可能多的操作交给客户端，这种方式也是 MongoDB 能够保持卓越性能的原因之一。随着业务的发展，MongoDB 也很有可能会遇到性能的瓶颈。

第 4 章
性能调优

4.1 机器负载高

现象：

大量慢查、数组查询、数据库 load、CPU 负载均衡非常高。

日志抓取：

```
2017-09-28T10:22:58.682+0800 I COMMAND [conn155426] command xxx.song command: find { find: "song", filter: { file.fds_path: "fds://bak/201709230/8589628/resources/SKA12208.wav" }, limit: 1, singleBatch: true } planSummary: COLLSCAN keysExamined:0 docsExamined:316840 cursorExhausted:1 numYields:2477 nreturned:1 reslen:1749 locks:{ Global: { acquireCount: { r: 4956 } }, Database: { acquireCount: { r: 2478 }, acquireWaitCount: { r: 1 }, timeAcquiringMicros: { r: 196 } }, Collection: { acquireCount: { r: 2478 } } } protocol:op_query 573ms
```

原因：

在日志中能够看出，查询计划是 COLLSCAN 全表扫描，效率是非常低的，因此添加索引对其进行优化。

处理办法：

从日志能看出，其对 file.fds_path 列进行了过滤，因此我们添加索引如下：

```
test:PRIMARY> db.song.createIndex({"file.fds_path":1},{background:true})
{
"createdCollectionAutomatically" : true,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
}
```

4.2 快速修改库名

背景：

业务建库不规范，导致库名需要重新定义，如何更快地修改库名呢？

解决办法：

通过改集合名实现改库名（限制：会阻塞所有 DB 操作，Shard 集群不能用，注意提前授权新库权限）。

命令实现：

```
var source = "competitive";########原库名
var dest = "product";##########新库名
var colls = db.getSiblingDB(source).getCollectionNames();
for (var i = 0; i < colls.length; i++) {
    var from = source + "." + colls[i];
    var to = dest + "." + colls[i];
    db.adminCommand({renameCollection: from, to: to});
}
```

插曲：

改完后 show dbs 变小了，这个统计没那么快，等一会儿再看就好了。接下来对文档数进行验证。

```
heleitest:PRIMARY> show dbs;
local              0.033GB
```

```
    competitive  0.029GB
    heleitest:PRIMARY> use competitive
    switched to db competitive
    heleitest:PRIMARY> show tables;
    system.profile
    t_ecommerce_comment
    t_ecommerce_goods
    t_keywordgroup
    t_media_crawlingtask
    t_media_info
    heleitest:PRIMARY> db.t_ecommerce_comment.count()
    167840
    heleitest:PRIMARY> db.t_ecommerce_goods.count()
    221
    heleitest:PRIMARY> db.t_keywordgroup.count()
    8
    heleitest:PRIMARY> db.t_media_crawlingtask.count()
    2
    heleitest:PRIMARY> db.t_media_info.count()
    11855
    heleitest:PRIMARY> var source = "competitive";
    heleitest:PRIMARY> var dest = "product";
    heleitest:PRIMARY>                var                colls                =
db.getSiblingDB(source).getCollectionNames();
    heleitest:PRIMARY> for (var i = 0; i < colls.length; i++) {
    ...     var from = source + "." + colls[i];
    ...     var to = dest + "." + colls[i];
    ...     db.adminCommand({renameCollection: from, to: to});
    ... }
    { "ok" : 1 }
    heleitest:PRIMARY> show dbs;
    local              0.033GB
    competitive  0.000GB
    product     0.002GB
    heleitest:PRIMARY> use competitive
    switched to db competitive
    heleitest:PRIMARY> show tables;
    system.profile
```

```
heleitest:PRIMARY> use product
switched to db product
heleitest:PRIMARY> show tables;
t_ecommerce_comment
t_ecommerce_goods
t_keywordgroup
t_media_crawlingtask
t_media_info
heleitest:PRIMARY> db.t_ecommerce_comment.count()
167840
heleitest:PRIMARY> db.t_ecommerce_goods.count()
221
heleitest:PRIMARY> db.t_keywordgroup.count()
8
heleitest:PRIMARY> db.t_media_crawlingtask.count()
2
```

备注：上述操作使用 count 进行粗略估算，如果需要精细估算，则可以使用 db.hash 命令。当然，从源头上杜绝这类改库名的需求，实现规范化上线才是根本。

4.3 dbhash 检查一致性

抽样检查 helei 库是否一致：

```
heleitest:PRIMARY> use helei
switched to db helei
heleitest:PRIMARY> db.runCommand( { dbHash: 1 } )
{
     "host" : "HE2:27020",
     "collections" : {
          "helei" : "364936adda4fb850f350829be59dc7af",
          "helei1" : "d44c35f746f1b53d9f01e652a47ec41a"
     },
     "md5" : "6692028b8f31b1941f8e194b584ef80f",
     "timeMillis" : 52,
     "fromCache" : [ ],
     "ok" : 1
```

```
}


heleitest:SECONDARY> use helei
switched to db helei
heleitest:SECONDARY> db.runCommand( { dbHash: 1 } )
{
    "host" : "HE1:27020",
    "collections" : {
        "helei" : "364936adda4fb850f350829be59dc7af",
        "helei1" : "d44c35f746f1b53d9f01e652a47ec41a"
    },
    "md5" : "6692028b8f31b1941f8e194b584ef80f",
    "timeMillis" : 65,
    "fromCache" : [ ],
    "ok" : 1
}
```

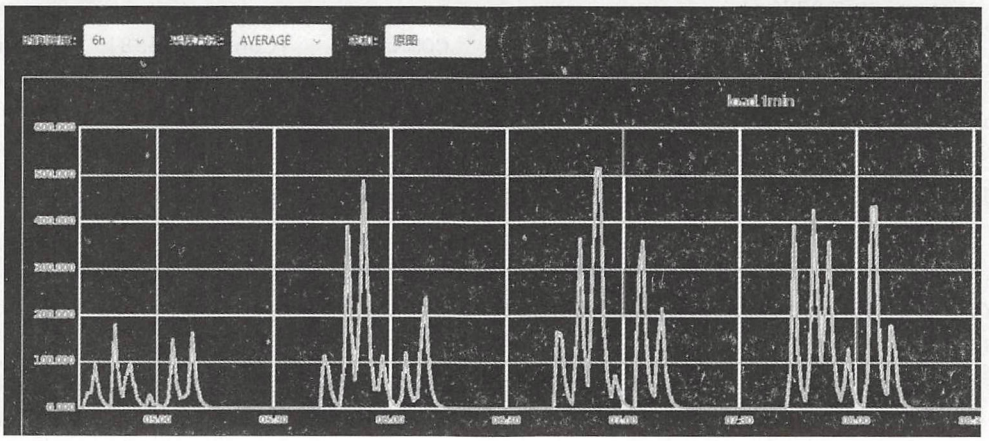
可以看到是一致的，没有问题，这个操作在大库上会非常耗时，且阻塞操作，需慎用。

4.4　使用索引却依旧性能低下

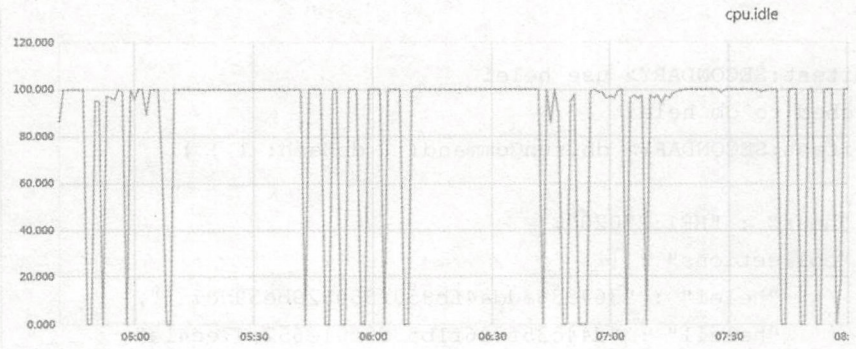
背景：

收到机器负载和 CPU 告警，发现 CPU 使用率很高，load 值也在几百上下。

排查过程如下图所示。



我们通过监控可以看到其负载很高。



其 CPU 也基本上时不时满负载运行。

MongoDB 日志：

```
    2018-03-29T09:45:51.782+0800    I    COMMAND    [conn4572921]    command
nlp_admin_db.BrainDictColl command: find { find: "BrainDictColl", filter:
{ domain: "baike", pr
    operty: "search", status: { $in: [ "ongoing" ] } }, sort: { _id: 1 }, skip:
710000, limit: 1000 } planSummary: IXSCAN { property: 1, domain: 1, status: 1,
_i
    d: 1 } cursorid:53850932942 keysExamined:710101 docsExamined:710101
numYields:5681 nreturned:101 reslen:24326 locks:{ Global: { acquireCount: { r:
11364 }, a
    cquireWaitCount: { r: 2 }, timeAcquiringMicros: { r: 61245 } }, Database:
{ acquireCount: { r: 5682 } }, Collection: { acquireCount: { r: 5682 } } } protocol
    :op_query 19871ms
    2018-03-29T09:45:51.789+0800    I    COMMAND    [conn4573658]    command
nlp_admin_db.BrainDictColl command: find { find: "BrainDictColl", filter:
{ domain: "baike", pr
    operty: "search", status: { $in: [ "ongoing" ] } }, sort: { _id: 1 }, skip:
673000, limit: 1000 } planSummary: IXSCAN { property: 1, domain: 1, status: 1,
_i
    d: 1 } cu rso rid:52484639023 keysExamined:673101 docsExamined:673101
numYields:5386 nreturned:101 reslen:24371 locks:{ Global: { acquireCount: { r:
10774 }, a
    cquireWaitCount: { r: 2 }, timeAcquiringMicros: { r: 93571 } }, Database:
{ acquireCount: { r: 5387 } }, Collection: { acquireCount: { r: 5387 } } } protocol
```

```
    :op_query 20617ms
    2018-03-29T09:45:51.794+0800    I    COMMAND    [conn4582654]    command
nlp_admin_db.BrainDictColl command: find { find: "BrainDictColl", filter:
{ domain: "baike", pr
    operty: "search", status: { $in: [ "ongoing" ] } }, sort: { _id: 1 }, skip:
708000, limit: 1000 } planSummary: IXSCAN { property: 1, domain: 1, status: 1,
_i
    d: 1 } cursorid:52133780984 keysExamined:708101 docsExamined:708101
numYields:5673 nreturned:101 reslen:24518 locks:{ Global: { acquireCount: { r:
11348 }, a
    cquireWaitCount: { r: 2 }, timeAcquiringMicros: { r: 24624 } }, Database:
{ acquireCount: { r: 5674 } }, Collection: { acquireCount: { r: 5674 } } } protocol
    :op_query 20863ms
    2018-03-29T09:45:51.795+0800    I    COMMAND    [conn4433255]    command
nlp_admin_db.BrainDictColl command: find { find: "BrainDictColl", filter:
{ domain: "baike", pr
    operty: "search", status: { $in: [ "ongoing" ] } }, sort: { _id: 1 }, skip:
685000, limit: 1000 } planSummary: IXSCAN { property: 1, domain: 1, status: 1,
_i
    d: 1 } cursorid:51946518105 keysExamined:685101 docsExamined:685101
numYields:5484 nreturned:101 reslen:24306 locks:{ Global: { acquireCount: { r:
10970 }, a
    cquireWaitCount: { r: 2 }, timeAcquiringMicros: { r: 25411 } }, Database:
{ acquireCount: { r: 5485 } }, Collection: { acquireCount: { r: 5485 } } } protocol
    :op_query 19419ms
    2018-03-29T09:45:51.796+0800    I    COMMAND    [conn4572915]    command
nlp_admin_db.BrainDictColl command: find { find: "BrainDictColl", filter:
{ domain: "baike", pr
    operty: "search", status: { $in: [ "ongoing" ] } }, sort: { _id: 1 }, skip:
672000, limit: 1000 } planSummary: IXSCAN { property: 1, domain: 1, status: 1,
_i
    d: 1 } cursorid:54510973417 keysExamined:672101 docsExamined:672101
numYields:5380 nreturned:101 reslen:24347 locks:{ Global: { acquireCount: { r:
10762 }, a
    cquireWaitCount: { r: 2 }, timeAcquiringMicros: { r: 20353 } }, Database:
{ acquireCount: { r: 5381 } }, Collection: { acquireCount: { r: 5381 } } } protocol
    :op query 19704ms
```

具体分析：

可以看到，这条查询在用了索引的情况下耗时为 4725ms。

```
sMongoMiliaoData:SECONDARY> db.runCommand(
...    {
...          explain: { find : "mycollection", filter : { domain :
"baike",property : "search",status : { $in : [ "ongoing" ] } }, sort : { _id :
1 }, skip : 709000, limit : 1000 },
...     verbosity: "executionStats"
...    }
... )
{
        "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "db.mycollection",
        "indexFilterSet" : false,
        "parsedQuery" : {
              "$and" : [
                    {
                          "domain" : {
                                "$eq" : "baike"
                          }
                    },
                    {
                         "property" : {
                               "$eq" : "search"
                          }
                    },
                    {
                         "status" : {
                               "$eq" : "ongoing"
                          }
                    }
              ]
        },
        "winningPlan" : {
             "stage" : "LIMIT",
```

```
                "limitAmount" : 1000,
                "inputStage" : {
                      "stage" : "SKIP",
                      "skipAmount" : 0,
                      "inputStage" : {
                            "stage" : "FETCH",
                            "inputStage" : {
                                  "stage" : "IXSCAN",
                                  "keyPattern" : {
                                       "property" : 1,
                                       "domain" : 1,
                                       "status" : 1,
                                       "_id" : 1
                                  },
                                  "indexName" : "property_1_domain_1_status
_1__id_1",
                                  "isMultiKey" : false,
                                  "multiKeyPaths" : {
                                        "property" : [ ],
                                        "domain" : [ ],
                                        "status" : [ ],
                                        "_id" : [ ]
                                  },
                                  "isUnique" : false,
                                  "isSparse" : false,
                                  "isPartial" : false,
                                  "indexVersion" : 1,
                                  "direction" : "forward",
                                  "indexBounds" : {
                                        "property" : [
                                              "[\"search\", \"search\"]"
                                        ],
                                        "domain" : [
                                            "[\"baike\", \"baike\"]"
                                        ],
                                        "status" : [
                                              "[\"ongoing\",
\"ongoing\"]"
```

```
                                        ],
                                        "_id" : [
                                                "[MinKey, MaxKey]"
                                        ]
                                    }
                                }
                            }
                        }
                    },
                    "rejectedPlans" : [
                        {
                            "stage" : "SKIP",
                                    ….
                                    ….
                                    ….
                                    ….
                                    "status" : [
                                        "[\"ongoing\", \"ongoing\"]"
                                    ]
                                }
                            }
                        }
                    }
                }
            ]
        },
        "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 1000,
            "executionTimeMillis" : 4725,
            "totalKeysExamined" : 710000,
            "totalDocsExamined" : 710000,
            "executionStages" : {
                "stage" : "LIMIT",
                "nReturned" : 1000,
                "executionTimeMillisEstimate" : 845,
                "works" : 710001,
```

```
"advanced" : 1000,
"needTime" : 709000,
"needYield" : 0,
"saveState" : 9708,
"restoreState" : 9708,
"isEOF" : 1,
"invalidates" : 0,
"limitAmount" : 1000,
"inputStage" : {
      "stage" : "SKIP",
      "nReturned" : 1000,
      "executionTimeMillisEstimate" : 845,
      "works" : 710000,
      "advanced" : 1000,
      "needTime" : 709000,
      "needYield" : 0,
      "saveState" : 9708,
      "restoreState" : 9708,
      "isEOF" : 0,
      "invalidates" : 0,
      "skipAmount" : 0,
      "inputStage" : {
            "stage" : "FETCH",
            "nReturned" : 710000,
            "executionTimeMillisEstimate" : 814,
            "works" : 710000,
            "advanced" : 710000,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 9708,
            "restoreState" : 9708,
            "isEOF" : 0,
            "invalidates" : 0,
            "docsExamined" : 710000,
            "alreadyHasObj" : 0,
            "inputStage" : {
                  "stage" : "IXSCAN",
                  "nReturned" : 710000,
```


```
                                "executionTimeMillisEstimate" : 503,
                                "works" : 710000,
                                "advanced" : 710000,
                                "needTime" : 0,

                                "needYield" : 0,
                                "saveState" : 9708,
                                "restoreState" : 9708,
                                "isEOF" : 0,
                                "invalidates" : 0,
                                "keyPattern" : {
                                     "property" : 1,
                                     "domain" : 1,
                                     "status" : 1,
                                     "_id" : 1
                                 },
                                "indexName"                                    :
"property_1_domain_1_status_1__id_1",
                                "isMultiKey" : false,
                                "multiKeyPaths" : {
                                      "property" : [ ],
                                      "domain" : [ ],
                                      "status" : [ ],
                                      "_id" : [ ]
                                 },
                                 "isUnique" : false,
                                 "isSparse" : false,
                                 "isPartial" : false,
                                 "indexVersion" : 1,
                                 "direction" : "forward",
                                 "indexBounds" : {
                                       "property" : [
                                            "[\"search\", \"search\"]"
                                        ],
                                        "domain" : [
                                             "[\"baike\", \"baike\"]"
                                        ],
                                       "status" : [
```

```
                                        "[\"ongoing\", \"ongoing\"]"
                                    ],
                                    "_id" : [
                                            "[MinKey, MaxKey]"
                                    ]
                                },
                                "keysExamined" : 710000,
                                "seeks" : 1,
                                "dupsTested" : 0,
                                "dupsDropped" : 0,
                                "seenInvalidated" : 0
                            }
                        }
                    }
            },
    "serverInfo" : {
            "host" : "c3-helei-mgdb02.bj",
            "port" : 27020,
            "version" : "3.4.4-1.4",
            "gitVersion" : "3ee0bb263c7a6747ac2cea4e54fa67eabc11faff"
    },
    "ok" : 1
```

通过对查询模型的分析，我们可以初步定位是因为 skip 的存在导致的效率低下。为了验证我们的猜想，尝试降低 skip 后面的数值，看一下耗时是否有提升。

如下所示，我们将 skip 改为 9000 后，查询速度明显提升了，用时为 955ms：

```
    rsMongoMiliaoData:SECONDARY> db.runCommand(    {      explain: { find :
"mycollection", filter : { domain : "baike",property : "search",status : { $in :
[ "ongoing" ] } }, sort : { _id : 1 }, skip : 9000, limit : 1000 },     verbosity:
"executionStats"    } )
    {
        "queryPlanner" : {
            "plannerVersion" : 1,
            "namespace" : "db.mycollection",
            "indexFilterSet" : false,
            "parsedQuery" : {
```

```
                "$and" : [
                        {
                                "domain" : {
                                        "$eq" : "baike"

                                }
                        },
                        {
                                "property" : {
                                        "$eq" : "search"
                                }
                        },
                        {
                                "status" : {
                                        "$eq" : "ongoing"
                                }
                        }
                ]
        },
        "winningPlan" : {
                "stage" : "LIMIT",
                "limitAmount" : 1000,
                "inputStage" : {
                        "stage" : "SKIP",
                        "skipAmount" : 0,
                        "inputStage" : {
                                "stage" : "FETCH",
                                "inputStage" : {
                                        "stage" : "IXSCAN",
                                        "keyPattern" : {
                                                "property" : 1,
                                                "domain" : 1,
                                                "status" : 1,
                                                "_id" : 1
                                        },
                                        "indexName" :
                                        "property_1_domain_1_status_1__id_1",
                                        "isMultiKey" : false,
```

```
                    "multiKeyPaths" : {
                          "property" : [ ],
                          "domain" : [ ],
                          "status" : [ ],
                          "_id" : [ ]
                     },
                     "isUnique" : false,
                     "isSparse" : false,
                     "isPartial" : false,
                     "indexVersion" : 1,
                     "direction" : "forward",
                     "indexBounds" : {
                           "property" : [
                                 "[\"search\", \"search\"]"
                            ],
                            "domain" : [
                                "[\"baike\", \"baike\"]"
                             ],
                             "status" : [
                                   "[\"ongoing\", \"ongoing\"]"
                              ],
                              "_id" : [
                                    "[MinKey, MaxKey]"
                                     ]
                               }
                      }
                }
          }
},
"rejectedPlans" : [
{
          "stage" : "SKIP",
          "skipAmount" : 9000,
          "inputStage" : {
                "stage" : "SORT",
                "sortPattern" : {
                      "_id" : 1
                            …..
```

```
                                        …..
                                        …..
                                        …..
                                        "isPartial" : false,

                                        "indexVersion" : 1,
                                        "direction" : "forward",
                                        "indexBounds" : {
                                              "domain" : [
                                                   "[\"baike\", \"baike\"]"
                                              ],
                                              "property" : [
                                                   "[\"search\", \"search\"]"
                                              ],
                                              "status" : [
                                                   "[\"ongoing\",
\"ongoing\"]"
                                              ]
                                        }
                                  }
                            }
                      }
                }
          ]
    },
    "executionStats" : {
         "executionSuccess" : true,
         "nReturned" : 1000,
         "executionTimeMillis" : 955,
         "totalKeysExamined" : 10000,
         "totalDocsExamined" : 10000,
         "executionStages" : {
              "stage" : "LIMIT",
              "nReturned" : 1000,
              "executionTimeMillisEstimate" : 71,
              "works" : 10001,
              "advanced" : 1000,
```

```
"needTime" : 9000,
"needYield" : 0,
"saveState" : 298,
"restoreState" : 298,
"isEOF" : 1,
"invalidates" : 0,
"limitAmount" : 1000,
"inputStage" : {
      "stage" : "SKIP",
      "nReturned" : 1000,
      "executionTimeMillisEstimate" : 61,
      "works" : 10000,
      "advanced" : 1000,
      "needTime" : 9000,
      "needYield" : 0,
      "saveState" : 298,
      "restoreState" : 298,
      "isEOF" : 0,
      "invalidates" : 0,
      "skipAmount" : 0,
      "inputStage" : {
            "stage" : "FETCH",
            "nReturned" : 10000,
            "executionTimeMillisEstimate" : 61,
            "works" : 10000,
            "advanced" : 10000,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 298,
            "restoreState" : 298,
            "isEOF" : 0,
            "invalidates" : 0,
            "docsExamined" : 10000,
            "alreadyHasObj" : 0,
            "inputStage" : {
                  "stage" : "IXSCAN",
                  "nReturned" : 10000,
                  "executionTimeMillisEstimate" : 31,
```

```
"works" : 10000,
"advanced" : 10000,
"needTime" : 0,
"needYield" : 0,
"saveState" : 298,
"restoreState" : 298,
"isEOF" : 0,
"invalidates" : 0,
"keyPattern" : {
      "property" : 1,
      "domain" : 1,
      "status" : 1,
      "_id" : 1
 },
 "indexName" :
 "property_1_domain_1_status_1__id_1",
 "isMultiKey" : false,
 "multiKeyPaths" : {
      "property" : [ ],
      "domain" : [ ],
      "status" : [ ],
      "_id" : [ ]
 },
"isUnique" : false,
"isSparse" : false,
"isPartial" : false,
"indexVersion" : 1,
"direction" : "forward",
"indexBounds" : {
      "property" : [
            "[\"search\", \"search\"]"
       ],
       "domain" : [
            "[\"baike\", \"baike\"]"
        ],
        "status" : [
             "[\"ongoing\", \"ongoing\"]"
         ],
         "_id" : [
```

```
                                                    "[MinKey, MaxKey]"
                                                ]
                                            },
                                            "keysExamined" : 10000,
                                            "seeks" : 1,
                                            "dupsTested" : 0,
                                            "dupsDropped" : 0,
                                            "seenInvalidated" : 0
                                        }
                                    }
                                }
                            }
    },
    "serverInfo" : {
            "host" : "c3-helei-mgdb02.bj",
            "port" : 27020,
            "version" : "3.4.4-1.4",
            "gitVersion" : "3ee0bb263c7a6747ac2cea4e54fa67eabc11faff"
     },
     "ok" : 1
    }
```

原因解释：

cursor.skip()方法通常很昂贵，因为它需要服务器从集合或索引的开始处遍历，以便在开始返回结果之前获取偏移或跳过位置。随着偏移量的增加，cursor.skip()的运行将变得更慢，并且CPU 密集度更高。对于较大的集合，cursor.skip()可能会成为 I/O 瓶颈。

在官方文档中的 skip 章节也有提及：

```
    Using Range Queries
    Range queries can use indexes to avoid scanning unwanted documents,
typically yielding better performance as the offset grows compared to
using cursor.skip() for pagination.
```

使用 db.myCollection.find().limit(5)查询效率较高；使用 db.myCollection.find(). skip(200000). limit(5)查询效率很低。

解决办法：

避免使用 skip。例如，如果有日期作为起点，则可以使用$gt 和 limit 及 sort 组合的方式来避免使用 skip。

```
    db.myCollection.find({created_date        :        {        $gt        :
max_created_date_from_last_result } }).limit(100).sort({created_date:true});
```

如果查询模型不能变更，则可以尝试添加缓存。

4.5 索引

一个集合最多只能有 64 个索引，前台创建一个索引单条最大为 100MB 内存，早期调用 createindexes 最高可用 6.4GB 内存。

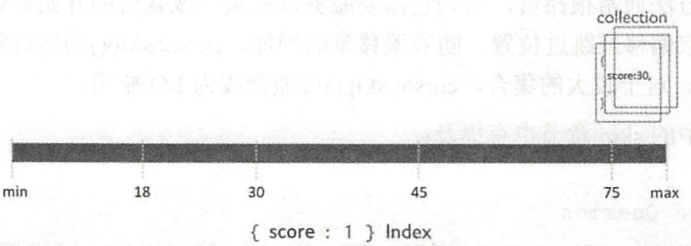
前台创建索引会阻塞一个库里的所有操作（read+write），但可以多个库同时在前台创建索引。

从 3.4 版本起有了 maxIndexBuildMemoryUsageMegabytes 参数，使用 createindexes 创建索引占用最大内存默认是 500MB，可以限制单条索引创建内存的大小。

4.5.1 单列索引

MongoDB 能够为文档集合中的任何字段提供索引功能。在默认情况下，所有集合在_id 字段上都有一个索引，应用程序和用户可以添加其他索引来支持重要的查询和操作。

下图描述单个字段上的升序/降序索引。



单个字段创建升序索引

使用如下命令为 records 集合的 score 字段创建一个升序索引：

```
    db.records.createIndex( { score: 1 } )
```

索引列后的字段值描述了该字段的索引类型。例如，值 1 指定按升序排列项目的索引。值 -1 指定按降序对项目进行排序的索引。对于单列索引而言，升序和降序都能够用到索引。

可以支持如下查询：

```
db.records.find( { score: 2 } )
db.records.find( { score: { $gt: 10 } } )
```

在嵌入式字段上创建索引

可以在嵌入文档中的字段上创建索引，就像可以索引文档中的最高级字段一样。嵌入式字段上的索引与嵌入式文档中的索引不同，嵌入式文档中的索引包括索引中嵌入文档最大索引大小的完整内容。相反，嵌入式字段上的索引允许使用“点符号”来指定索引的嵌入式文档。

例如，一个名为 records 的集合，该集合包含类似以下示例的文档。

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

以下命令为 records 集合的 location 字段里的 state 嵌入文档创建了索引。

```
db.records.createIndex( { "location.state": 1 } )
```

上述索引在如下查询中被用到：

```
db.rìords.find( { "location.state": "CA" } )
db.records.find( { "location.city": "Albany", "location.state": "NY" } )
```

在嵌入式文档上创建索引

例如，一个名为 records 的集合，该集合包含类似以下示例的文档。

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

location 字段是嵌入式文档，包含嵌入的字段 state 和 city。以下命令在整个 location 字段上

创建一个索引。

```
db.records.createIndex( { location: 1 } )
```

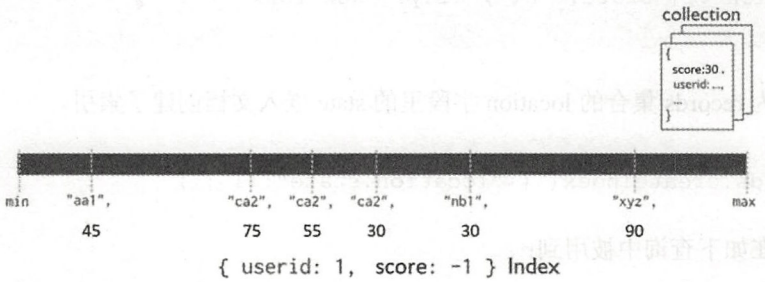
上述索引在如下查询中被用到：

```
db.records.find( { location: { city: "New York", state: "NY" } } )
```

虽然查询可以使用索引，但结果集不包含上述示例文档。在嵌入式文档上执行相应匹配命令时，字段顺序和嵌入式文档必须完全匹配。

4.5.2 复合索引

MongoDB 支持复合索引，其中单个索引结构持有对集合文档中多个字段的引用。下图说明了两个字段上的复合索引示例。



在 MongoDB 中，复合索引最多可以有 31 列。

复合索引可以支持匹配多个字段的查询。

创建复合索引

复合索引不支持 Hash 索引。

例如，有一个 products 集合内容如下：

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
```

```
 "type": "cases"
}
```

使用如下命令在 products 集合的 item 和 stock 字段上创建升序索引：

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

复合索引中列出的字段顺序很重要。索引会使得集合先根据 item 进行排序，并且按照 item 值进行升序排序，如果 item 相同，则继续按照 stock 值进行升序排序。

除了支持在所有索引字段上匹配的查询，复合索引还可以支持与索引字段的前缀相匹配的查询。也就是说，索引支持对 item 字段及 item 和 sotck 字段的查询：

```
db.products.find( { item: "Banana" } )
db.products.find( { item: "Banana", stock: { $gt: 5 } } )
```

复合索引的排序

索引以升序（1）或降序（-1）对字段内容进行排序。对于单字段索引，键的排序顺序并不重要，因为 MongoDB 可以在任一方向上遍历索引。但是对于复合索引，在不确定索引是否可以支持排序操作时，排序顺序可能很重要。

例如，一个集合包含字段 username 和 date 的文档，应用程序可以发出查询，返回的结果首先按 username 值进行升序排序，然后按 date 值进行降序（即最近到最后）排序。比如：

```
db.events.find().sort( { username: 1, date: -1 } )
```

或者返回结果的查询先按 username 值进行降序排序，然后按 date 值进行升序排序，例如：

```
db.events.find().sort( { username: -1, date: 1 } )
```

针对上述索引，只要创建一个索引就能够支持：

```
db.events.createIndex( { "username" : 1, "date" : -1 } )
```

需要值得注意的是，上述索引并不支持类似这样的查询：

```
db.events.find().sort( { username: 1, date: 1 } ) 和
db.events.find().sort( { username: -1, date: -1 } )
```

这样的查询需要创建索引：

```
db.events.createIndex( { "username" : 1, "date" : 1 } )
```

关于索引的排序可以简记为：单列索引正反向排序都不受影响，复合索引则是乘以（–1）的排序可以用相同的索引，即 1，1 和–1，–1 可以使用相同索引，–1，1 和 1，–1 可以使用相同的索引。

最左前缀原则

例如，如下索引：

```
{ "item": 1, "location": 1, "stock": 1 }
```

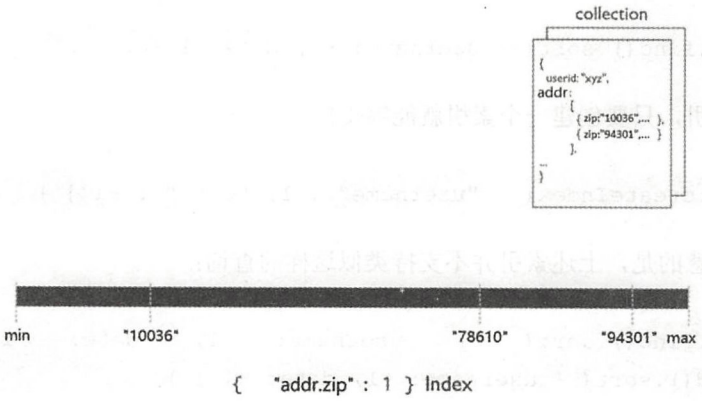
MongoDB 能够针对如下查询使用索引：

- item 字段；
- item 字段和 location 字段；
- tem 字段和 location 字段及 stock 字段。

4.5.3 多键索引

要索引一个包含数组值的字段，MongoDB 会为数组中的每个元素创建一个索引键。这些多键索引支持针对数组字段的高效查询，多键索引可以在包含标量值（例如，字符串、数字）和嵌套文档的数组上构建。

如下图所示，对文档创建多键索引 addr.zip，且支持数组字段。



标量值是指既不是嵌入式文档，也不是数组的值。

如果任何索引字段是数组，则 MongoDB 会自动创建一个多键索引，不需要明确指定多键类型。

从 MongoDB 3.4 开始，对于使用 MongoDB 3.4 或更高版本创建的多键索引，MongoDB 会跟踪那些会导致索引成为多键索引的索引字段，跟踪这些信息可以让 MongoDB 查询引擎使用更窄的索引界限。

索引界限

如果索引是多键，那么索引边界的计算要遵循特殊规则，这一规则后面会讲到。

多键唯一索引

对于唯一索引，唯一约束适用于集合中的单个文档而不是一个文档中数组的唯一。

由于唯一约束适用于单独的文档，对于多键唯一索引，只要文档的索引键值不重复，则对于多键唯一索引，文档可能具有导致重复索引键值的数组元素。

```
    heleitest:PRIMARY> db.helei.find()
    { "_id" : 1, "a" : [ { "loc" : "A", "qty" : 5 }, { "qty" : 10 } ] }
    { "_id" : 2, "a" : [ { "loc" : "A" }, { "qty" : 5 } ] }
    { "_id" : 3, "a" : [ { "loc" : "A", "qty" : 10 } ] }
    heleitest:PRIMARY> db.helei.createIndex( { "a.loc": 1, "a.qty": 1 }, { unique:
true } )
    {
            "createdCollectionAutomatically" : false,
            "numIndexesBefore" : 1,
            "numIndexesAfter" : 2,
            "ok" : 1
    }
    heleitest:PRIMARY> db.helei.insert( { _id: 4, a: [ { loc: "B" }, { loc:
"B" } ] } )
    WriteResult({ "nInserted" : 1 })
```

对于唯一索引，如果索引列缺失，则会为其配置 null 值，如果有两个文档的索引列都缺失，则 null 之间会重复、抛错：

```
    heleitest:PRIMARY> db.helei.insert({ _id: 5, a: [ { loc: "B"}, {qty: null} ] })
    WriteResult({
```

```
        "nInserted" : 0,
        "writeError" : {
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: helei.helei
index: a.loc_1_a.qty_1 dup key: { : \"B\", : null }"
        }
    })
    heleitest:PRIMARY> db.helei.insert({ _id: 5, a: [ { loc: "B"} ] })
    WriteResult({
        "nInserted" : 0,
        "writeError" : {
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: helei.helei
index: a.loc_1_a.qty_1 dup key: { : \"B\", : null }"
        }
    })
```

限制

如果存在多个数组，则无法对其创建复合索引，例如：

```
{ _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }
```

针对上述集合，无法对其创建索引{ a: 1, b: 1 }，因为 a 和 b 都是数组：

```
heleitest:PRIMARY> db.t.createIndex( { "a": 1, "b": 1 })
{
"ok" : 0,
"errmsg" : "cannot index parallel arrays [b] [a]",
"code" : 10088
}
```

反过来，如果已经存在索引{ a: 1, b: 1 }，那么集合不能插入 a 和 b 都是数组的文档，而如下的内容则不受影响：

```
{ _id: 1, a: [1, 2], b: 1, category: "A array" }
{ _id: 2, a: 1, b: [1, 2], category: "B array" }
```

针对数组：

```
{ _id: 1, a: [ { x: 5, z: [ 1, 2 ] }, { z: [ 1, 2 ] } ] }
{ _id: 2, a: [ { x: 5 }, { z: 4 } ] }
```

我们可以针对嵌入式字段创建索引{ "a.x": 1, "a.z": 1 }，当然这也要求集合中的文档不能同时都是数组。

分片

不能指定一个多键索引作为分片键索引。

哈希索引

哈希索引不可以是多键索引。

覆盖查询

MongoDB 不支持对数组的多键索引覆盖查询。

在数组上的查询表现

当一个查询为一个数组指定完全匹配时，MongoDB 可以使用多键索引来查询数组的第一个元素，但不能使用多键索引扫描来查找整个数组。相反，在使用多键索引查找数组的第一个元素之后，MongoDB 会检索相关文档并筛选其数组与查询中的数组相匹配的文档。

inventory 集合中有如下内容：

```
heleitest:PRIMARY> db.inventory.find()
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
{ "_id" : 8, "type" : "food", "item" : "ddd", "ratings" : [ 9, 5 ] }
{ "_id" : 9, "type" : "food", "item" : "eee", "ratings" : [ 5, 9, 5 ] }
```

我们在 ratings 上创建一个索引：

```
heleitest:PRIMARY> db.inventory.createIndex( { ratings: 1 } )
{
"createdCollectionAutomatically" : false,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
```

```
"ok" : 1
}
```

查询

```
heleitest:PRIMARY> db.inventory.find( { ratings: [ 5, 9 ] } )
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
heleitest:PRIMARY>
```

MongoDB 可以使用多键索引来查找数组中任何位置有 5 的文档。然后，MongoDB 检索这些文档并筛选其数组等于查询数组[5, 9]的文档。

数组子文档查询

在 inventory 集合中有如下内容：

```
heleitest:PRIMARY> db.inventory.find()
{ "_id" : 1, "item" : "abc", "stock" : [ { "size" : "S", "color" : "red",
"quantity" : 25 }, { "size" : "S", "color" : "blue", "quantity" : 10 }, { "size" :
"M", "color" : "blue", "quantity" : 50 } ] }
```

对 stock.size 和 stock.quantity 创建索引：

```
db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
```

这样的索引对要符合“最左前缀原则”的查询，例如：

```
db.inventory.find( { "stock.size": "M" } )
db.inventory.find( { "stock.size": "S", "stock.quantity": { $gt: 20 } } )
```

都能够使用该索引。

这个索引还能够支持 sort 排序操作，比如如下查询：

```
db.inventory.find( ).sort( { "stock.size": 1, "stock.quantity": 1 } )
db.inventory.find( { "stock.size": "M" } ).sort( { "stock.quantity": 1 } )
```

多键索引界限

survey 集合中有如下内容：

```
heleitest:PRIMARY> db.survey.find()
{ "_id" : 1, "item" : "ABC", "ratings" : [ 2, 9 ] }
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
{ "_id" : 3, "item" : "ZZZ", "ratings" : [ 1, 2 ] }
```

创建如下索引：

```
db.survey.createIndex( { ratings: 1 } )
```

以下查询使用$elemMatch 来要求数组至少包含一个匹配两个条件的单个元素：

```
heleitest:PRIMARY> db.survey.find( { ratings : { $elemMatch: { $gte: 3, $lte:
6 } } } )
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
```

如果查询未使用$elemMatch 条件，则 MongoDB 不能与多键索引边界相交，比如以下查询：

```
heleitest:PRIMARY> db.survey.find( { ratings : { $gte: 3, $lte: 6 } } )
{ "_id" : 1, "item" : "ABC", "ratings" : [ 2, 9 ] }
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
```

该查询将在数组中搜索至少一个大于或等于 3 的元素和至少一个小于或等于 6 的元素。由于单个元素不需要满足两个条件，所以 MongoDB 不会与边界相交，因此将两个结果都查询出来了。

复合索引边界是指对复合索引的多个键使用边界。例如，给定[3，Infinity]的字段 a 上的界限和[-Infinity，6]的字段 b 上的界限的复合索引{a：1，b：1}，复合边界导致使用两个边界。

上述集合创建如下索引：

```
heleitest:PRIMARY> db.survey.createIndex( { item: 1, ratings: 1 } )
heleitest:PRIMARY> db.survey.find( { item: "XYZ", ratings: { $gte: 3 } } )
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
heleitest:PRIMARY> db.survey.find( {
...    item: { $gte: "L", $lte: "Z"}, ratings : { $elemMatch: { $gte: 3, $lte:
6 } }
... } )
{ "_id" : 2, "item" : "XYZ", "ratings" : [ 4, 3 ] }
```

4.5.4 文本索引

一个集合最多只能创建一个文本索引。

使用如下方式对 reviews 集合 comments 字段创建一个文本索引：

```
db.reviews.createIndex( { comments: "text" } )
```

可以对多个字段创建文本索引：

```
db.reviews.createIndex(
   {
     subject: "text",
     comments: "text"
   }
 )
```

4.5.5 2dsphere 索引

2dsphere 索引支持球面几何查询。2dsphere index 支持所有 MongoDB 地理空间查询：包含交集和邻近的查询。2dsphere 索引支持以 GeoJSON 对象和传统坐标对存储的数据。对于传统坐标对，索引将数据转换为 GeoJSON Point。

版本 2 和更高版本的 2dsphere 索引总是稀疏并忽略稀疏选项。如果文档缺少 2dsphere 索引字段（或者该字段为 null 或空数组），则 MongoDB 不会将该文档的条目添加到索引。早期版本的 MongoDB 只支持 2dsphere（版本 1）索引。2dsphere（版本 1）索引默认情况下不是稀疏的，并且会拒绝具有空位置字段的文档。

geoNear 和$ geoNear 限制

geoNear 命令和$ geoNear 要求集合至多只有一个 2dsphere/2d 索引，而地理空间查询运算符（例如，$ near 和$ geoWithin）允许集合具有多个地理空间索引。

geoNear 命令和$ geoNear 管线阶段的地理空间索引限制存在，因为 geoNear 命令和$ geoNear 都不包含位置字段。因此，多个 2d 索引或 2dsphere 索引之间的索引选择是不允许的。

地理空间查询运算符没有这样的限制，因为这些运算符都有位置字段。

shard key 限制

对集合进行分片时，不能将 2dsphere 索引用作分片键。但是可以使用其余的字段作为分片

键，在分片集合上创建地理空间索引。

创建一个 2dsphere 索引：

```
db.collection.createIndex({<location field>: "2dsphere"})
```

places 集合有如下内容：

```
heleitest:PRIMARY> db.places.find()
{ "_id" : ObjectId("5aa8e363eda0da2c17cd53ef"), "loc" : { "type" : "Point", "coordinates" : [ -73.97, 40.77 ] }, "name" : "Central Park", "category" : "Parks" }
{ "_id" : ObjectId("5aa8e364eda0da2c17cd53f0"), "loc" : { "type" : "Point", "coordinates" : [ -73.88, 40.78 ] }, "name" : "La Guardia Airport", "category" : "Airport" }
```

loc 创建一个 2dsphere 索引：

```
heleitest:PRIMARY> db.places.createIndex( { loc : "2dsphere" } )
```

创建一个复合索引：

```
heleitest:PRIMARY> db.places.createIndex( { loc : "2dsphere" , category : -1, name: 1 } )
```

与 2d 索引不同，2dsphere 索引不要求位置列在第一个：

```
heleitest:PRIMARY> db.places.createIndex( { category : 1 , loc : "2dsphere" } )
```

4.5.6 2d 索引

对二维平面上存储的数据使用 2d 索引。2d 索引适用于 MongoDB 2.2 及更早版本中使用的传统坐标。

在以下情况下使用 2d 索引：

数据库具有来自 MongoDB 2.2 或更早版本的遗留传统坐标对，并且不打算将任何位置数据存储为 GeoJSON 对象。

注意事项：

geoNear 和$geoNear 限制

geoNear 命令和$geoNear 要求集合至多只有一个 2dsphere/2d 索引，而地理空间查询运算符（例如，$near 和$geoWithin）允许集合具有多个地理空间索引。

geoNear 命令和$geoNear 管线阶段的地理空间索引限制存在，因为 geoNear 命令和$geoNear 都不包含位置字段。因此，多个 2d 索引或 2dsphere 索引之间的索引选择是不允许的。

地理空间查询运算符没有这样的限制，因为这些运算符都有位置字段。

在分割集合时，不能将 2d 索引用作分片键。但是，可以使用不同的字段作为分片键，在分片集合上创建地理空间索引。

表现

2d 索引支持平面计算。2d 索引还支持球体上的$nearSphere 计算，但是对于球体上的几何计算（例如$ geoWithin），就要将数据存储为 GeoJSON 对象并使用 2dsphere 索引。

2d 索引可以引用两个字段，第一个必须是位置字段。二维复合索引构造查询，首先在位置字段中选择，然后通过附加条件过滤这些结果。复合 2d 索引可以涵盖查询。

稀疏

2d 索引总是稀疏并忽略稀疏选项。如果文档缺少 2d 索引字段（或者该字段为 null 或空数组），则 MongoDB 不会将文档的条目添加到 2d 索引中。

对于包含 2d 索引键和其他类型键的复合索引，只有 2d 索引字段确定索引是否引用文档。

4.5.7 Hash 索引

哈希索引通过索引字段的哈希值来维护条目。

哈希索引支持使用哈希分片键的分片，基于哈希的分片使用字段的哈希索引作为分片键来区分分片数据。

使用哈希分片键对集合进行分片会导致数据分布更加随机和均匀。

哈希索引使用散列函数来计算索引字段的哈希值。哈希索引不支持多键（即数组）索引。

创建哈希索引：

```
db.collection.createIndex( { _id: "hashed" } )
```

不可以创建具有哈希索引字段的复合索引，或者对哈希索引指定唯一约束。但是，可以在一个字段上同时创建哈希索引和升序/降序（即非哈希）索引。

4.5.8 一条 SQL 创建多个索引

一条命令创建多个索引，为 billbear 数据库的 t_act 集合创建多个索引：

```
db.getSiblingDB("billbear").runCommand(
  {
    createIndexes: "t_act",
    indexes: [
        {
            key: {
                banks:1,
                order:1,
                _id:-1
            },
name: "banks1_order1_id-1",
            unique: true,
background:true

        },
        {
            key: {
                usage_scene:1,
                order:1,
                _id:-1
            },
name: "scene1_order1_id-1",
            unique: true,
background:true
        },
        {
            key: {
                available_day:1,
                order:1,
                _id:-1
            },
name: "avaday1_order1_id-1",
            unique: true,
background:true
```

```
        },
{
            key: {
                cities:1,
                order:1,
                _id:-1
            },
name: "cities1_order1_id-1",
            unique: true,
background:true
        },
{
            key: {
                special_weekdays:1,
                order:1,
                _id:-1
            },
name: "spweek1_order1_id-1",
            unique: true,
background:true
        }
    ]
  }
)
```

4.6 索引属性

4.6.1 TTL 索引

TTL 索引是特殊的单字段索引，MongoDB 可以在特定的时间或特定的时间段后使用它自动从集合中删除文档。数据过期对于某些类型的信息非常有用，例如，机器生成的事件数据、日志和会话信息，这些信息只需要在有限的时间内保留在数据库中。

要创建 TTL 索引，可使用带有 expireAfterSeconds 选项的 db.collection.createIndex()方法，该方法的值为日期或包含日期值的数组。

例如，要在 eventlog 集合的 lastModifiedDate 字段上创建 TTL 索引，可在 mongo Shell 中使

用以下操作：

```
    heleitest:PRIMARY> db.eventlog.createIndex( { "lastModifiedDate": 1 },
{ expireAfterSeconds: 3600 } )
```

TTL 索引的适用场景如下。

数据到期

TTL 索引在索引字段值后超过指定秒数后过期，即到期阈值是索引字段值加上指定的秒数。

如果该字段是一个数组，并且索引中有多个日期值，则 MongoDB 会使用数组中最低（即最早）的日期值来计算到期阈值。

如果文档中的索引字段不是日期或包含日期值的数组，则文档不会过期。

如果文档不包含索引字段，则文档不会过期。

删除操作

mongod 中的后台线程读取索引中的值并从集合中删除过期的文档。

当 TTL 线程处于活动状态时，你将在 db.currentOp()的输出或由数据库分析器收集的数据中看到删除操作。

删除操作的时间

在后台构建 TTL 索引时，TTL 线程可以在构建索引时开始删除文档。如果在前台构建 TTL 索引，则 MongoDB 在索引构建完成后立即开始删除过期的文档。

TTL 索引不保证过期的数据将在到期后立即删除。文档到期时间和 MongoDB 从数据库中删除文档的时间之间可能存在延迟。

移除过期文档的后台任务每 60 秒运行一次。因此，在文档到期和后台任务运行期间，文档可能会保留在一个集合中。

因为删除操作的持续时间取决于 mongod 实例的工作负载，过期数据可能会存在超过 60s 的时间。

副本集

在副本集成员上，只有当成员处于 Primary 状态时，TTL 后台线程会删除文档。成员处于 secondary 状态时，TTL 后台线程处于空闲状态。Secondary 成员从 Primary 节点复制删除操作。

限制

TTL 索引是单字段索引，复合索引不支持 TTL 并忽略 expireAfterSeconds 选项。

_id 字段不支持 TTL 索引。

无法在 capped 集合上创建 TTL 索引，因为 MongoDB 无法从 capped 集合中移除文档。

不能使用 createIndex()更改现有索引 expireAfterSeconds 的值，可以使用 collMod 数据库命令来变更过期时间，也可以先删除索引再重新按需要创建。

如果字段中已存在非 TTL 单字段索引，则不能在同一字段上创建 TTL 索引，要将非 TTL 单字段索引更改为 TTL 索引，必须首先删除索引并使用 expireAfterSeconds 选项重新创建。

4.6.2 唯一索引

唯一索引确保索引字段不存储重复值，即强制索引字段的唯一性。在默认情况下，MongoDB 在创建集合时在_id 字段上创建唯一索引。

创建唯一索引

以下命令对 members 集合的 user_id 列创建了一个单列唯一索引：

```
    db.members.createIndex( { "user_id": 1 }, { unique: true } )
```

以下命令对 members 集合的 groupNumber、lastname、firstname 创建了一个复合唯一索引

```
    db.members.createIndex( { groupNumber: 1, lastname: 1, firstname: 1 },
{ unique: true } )
```

例如，我们对 heleiuniq 集合插入数据并创建唯一索引：

```
    heleitest:PRIMARY> db.heleiuniq.insert({ _id: 1, a: [ { loc: "A", qty: 5 },
{ qty: 10 } ] })
    WriteResult({ "nInserted" : 1 })
    heleitest:PRIMARY> db.heleiuniq.createIndex( { "a.loc": 1, "a.qty": 1 },
{ unique: true } )
    heleitest:PRIMARY> db.heleiuniq.insert( { _id: 2, a: [ { loc: "A" }, { qty:
5 } ] } )
    WriteResult({ "nInserted" : 1 })
    heleitest:PRIMARY> db.heleiuniq.insert( { _id: 3, a: [ { loc: "A", qty:
10 } ] } )
    WriteResult({ "nInserted" : 1 })
```

可以看出，如果再插入一个只包含 loc 为 A、qty 是 null 的数据时，则会报出重复的错误：

```
heleitest:PRIMARY> db.heleiuniq.insert( { _id: 4, a: [ { loc: "A" }, { qty:
100 } ] } )
WriteResult({
"nInserted" : 0,
"writeError" : {
"code" : 11000,
"errmsg" : "E11000 duplicate key error collection: helei.heleiuniq index:
a.loc_1_a.qty_1 dup key: { : \"A\", : null }"
}
})
```

限制

如果集合已经包含违反索引唯一约束的数据，则 MongoDB 无法在指定的索引字段上创建唯一索引。

不能在 Hash 索引上指定唯一约束。

唯一索引对缺失列的处理

如果文档在唯一索引中没有索引字段的值，则索引将为此文档存储 null 值。由于唯一的约束，MongoDB 将只允许一个缺少索引字段的文档。如果有多个文档没有索引字段的值或缺少索引字段，则在添加唯一索引时将失败，并报出重复键错误。

4.6.3 部分索引

部分索引仅索引符合指定过滤器表达式的集合中的文档。通过索引集合中的文档子集，部分索引的创建和维护的存储需求更低，性能成本也更低。

创建部分索引

要创建部分索引，可使用带 partialFilterExpression 选项的 db.collection.createIndex()方法。partialFilterExpression 选项接收一个指定过滤条件的文档，例如：

- 相等表达式（即 file、value 或使用$eq 运算符）；
- $ exists 表达式；
- $ gt、$ gte、$ lt、$ lte 表达式；

- $type 表达式；
- $and。

对 restaurants 的 rating 大于 5 的文档创建 cuisine 和 name 的复合索引：

```
heleitest:PRIMARY> db.restaurants.createIndex(
...    { cuisine: 1, name: 1 },
...    { partialFilterExpression: { rating: { $gt: 5 } } }
... )
{
"createdCollectionAutomatically" : true,
"numIndexesBefore" : 1,
"numIndexesAfter" : 2,
"ok" : 1
}
```

限制

在 MongoDB 中，不能创建仅在选项上有所不同的索引的多个版本。因此，不能创建仅是过滤器表达式不同的多个部分索引。

不能指定 partialFilterExpression 选项和稀疏选项。

早期版本的 MongoDB 不支持部分索引，对于分片集群或副本集合，所有节点必须是 3.2 版本。

_id 索引不可以是部分索引。

shard key 索引不可以是部分索引。

4.6.4 稀疏索引

即使索引字段包含空值，稀疏索引也只包含具有索引字段文档的条目，索引跳过任何缺少索引字段的文档。索引是“稀疏”的，因为它不包含集合的所有文档。相比之下，非稀疏索引包含集合中的所有文档，为那些不包含索引字段的文档存储空值。

从 MongoDB 3.2 版开始，MongoDB 提供了创建部分索引的选项，部分索引提供了稀疏索引功能的超集。如果使用的是 MongoDB 3.2 版或更高版本，则部分索引应优先于稀疏索引。

创建稀疏索引

使用如下语句对 addresses 集合 xmpp_id 列创建一个稀疏索引：

```
heleitest:PRIMARY> db.addresses.createIndex( { "xmpp_id": 1 }, { sparse:
true } )
```

稀疏索引将不会对不包含 xmpp_id 列的文档进行索引。

不完整结果

如果稀疏索引会导致查询和排序操作的结果集不完整，则 MongoDB 将不会使用该索引，除非 hint()明确指定索引。

例如，查询{x：{$ exists：false}}不会在 x 字段上使用稀疏索引，除非强制执行索引。

稀疏复合索引

只包含升序/降序索引键的稀疏复合索引及索引文档，只要文档至少包含一个键即可。

对于包含地理空间键（即 2dsphere、2d 或 geoHaystack 索引键）及升序/降序索引键的稀疏复合索引，只有文档中存在地理空间字段时才能确定索引是否可以引用文件。

对于包含文本索引键及升序/降序索引键的稀疏复合索引，只有文本索引字段的存在才能确定索引是否引用文档。

稀疏唯一索引

既稀疏又唯一的索引可防止文档具有重复值的字段，但允许多个文档忽略该键。

4.7 在大集合上创建索引

在默认情况下，在大集合上创建索引会阻止该数据库上的所有其他操作。

使用如下命令在后台创建索引：

```
db.people.createIndex( { zipcode: 1 }, { background: true } )
```

在后台创建索引不会阻塞其他操作，因此无论是大集合还是小集合，都建议使用后台创建索引，以养成习惯。

后台创建索引也可以和其余选项组合使用，例如：

```
heleitest:PRIMARY> db.people.createIndex( { zipcode: 1 }, { background: true,
sparse: true } )
```

尽量在后台创建索引，以便在创建索引时可以运行其他数据库操作。但是在建立索引之前，

mongo Shell 会话或连接将会阻塞，直到索引构建完成。要继续向数据库发出命令，可打开另一个连接或 mongo 实例。

查询不会使用正在创建的索引，索引只有在索引构建完成后才可用。

性能表现

后台索引操作使用比正常“前台”索引构建更慢的增量方法。如果索引需要的 RAM 大小大于剩余的 RAM，那么增量过程可能比前台构建花费更长的时间。

构建索引可能会严重影响数据库的性能。如果可能的话，则在指定维护时段内建立索引。

从 MongoDB 3.4 版起，可以使用数据库命令 createIndexes 在集合上构建一个或多个索引，createIndexes 操作的默认内存使用限制是 500MB。可以通过设置 maxIndexBuildMemoryUsageMegabytes 服务器参数来覆盖此限制。

createIndexes 在磁盘上使用内存和临时文件组合来完成索引构建。达到内存限制后，createIndexes 会在--dbpath 目录内名为_tmp 的子目录中使用临时磁盘文件，以获取其他临时空间。设置的内存限制越高，索引构建就可以完成得越快。但注意不要将此限制设置得比可用内存高太多，否则系统可能会耗尽可用内存。

创建索引时中断

如果在 mongod 进程终止时正在进行后台索引构建，则当实例重新启动时，索引构建将作为前台索引构建重新启动。如果索引构建遇到任何错误（例如，重复键错误），则 mongod 将退出且出现错误。

要在失败的索引构建后启动 mongod，可在启动时使用 storage.indexBuildRetry 或 --noIndexBuildRetry 跳过索引构建。

4.8 索引交集

索引交集不让复合索引变得没用。因为如果要使用复合索引，则需要按照“最左前缀”原则来进行索引，而索引交集则不存在这个问题。

例如，如果收集订单具有以下索引：

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
```

则 MongoDB 依旧能够使用索引交集：

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } )
```

复合索引的索引交集

对于如下索引：

```
{ status: 1, ord_date: -1 }
```

MongoDB 能够支持该索引的如下查询：

```
db.orders.find( { status: { $in: ["A", "P" ] } } )
db.orders.find(
   {
     ord_date: { $gt: new Date("2014-02-01") },
     status: {$in:[ "P", "A" ] }
   }
)
```

但不支持以下查询：

```
db.orders.find( { ord_date: { $gt: new Date("2014-02-01") } } )
db.orders.find( { } ).sort( { ord_date: 1 } )
```

但是，如果将其拆为两个索引：

```
{ status: 1 }
{ ord_date: -1 }
```

则这两个索引可以单独或通过索引交叉来支持上述 4 个查询。

创建单列索引或复合索引之间的选择取决于系统的具体情况。

索引交集与 sort

例如，集合中有如下索引：

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
{ status: 1 }
{ ord_date: -1 }
```

此时，这类查询就不能够使用索引进行排序：

```
db.orders.find( { qty: { $gt: 10 } } ).sort( { status: 1 } )
```

也就是说，MongoDB 不会将 qty 索引用于查询，status 或{status:1,ord_date:-1}索引用于排序。但是对于下述查询，由于{ status: 1, ord_date: -1 }索引覆盖了部分查询，因此可以用到索引排序：

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } ).sort( { ord_date: -1 } )
```

4.9 索引排序

在 MongoDB 中，排序操作可以通过基于索引中的排序检索文档来获取排序顺序。如果查询计划程序无法从索引获取排序顺序，则会将结果在内存中排序。使用索引的排序操作通常比没有使用索引的排序操作有更好的性能。另外，不使用索引的排序操作在使用 32 MB 内存时将中止。

由于对 MongoDB 3.6 中数组字段的排序行为的更改，在对使用多键索引的数组进行排序时，查询计划包含阻塞 SORT 阶段。新的分类行为可能会对性能产生负面影响。

如果在单个字段上有上升或下降索引，则字段上的排序操作可以在任一方向上进行。

例如，为集合 records 的字段 a 创建一个升序索引：

```
db.records.createIndex( { a: 1 } )
```

这个索引可以支持两个 sort：

```
db.records.find().sort( { a: 1 } )
db.records.find().sort( { a: -1 } )
```

多列排序

可以指定索引的所有键或子集上的排序，但是排序键必须按照它们在索引中出现的顺序列出。例如，索引键模式{a：1，b：1}可以支持在{a：1，b：1}上排序，但不支持在{b：1，a：1}上排序。

对于使用复合索引进行排序的查询，所有键的指定排序方向必须与索引键模式匹配或匹配索引键模式的倒数。例如，索引键模式{a：1，b：-1}可以支持{a：1，b：-1}和{a：-1，b：1}

上的排序，但不支持{a： -1，b：-1}或{a：1，b：1}。

最左前缀

例如，data集合创建如下索引：

```
db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )
```

下表解释了符合最左前缀原则的排序。

Example	Index Prefix
db.data.find().sort({ a: 1 })	{ a: 1 }
db.data.find().sort({ a: -1 })	{ a: 1 }
db.data.find().sort({ a: 1, b: 1 })	{ a: 1, b: 1 }
db.data.find().sort({ a: -1, b: -1 })	{ a: 1, b: 1 }
db.data.find().sort({ a: 1, b: 1, c: 1 })	{ a: 1, b: 1, c: 1 }
db.data.find({ a: { $gt: 4 } }).sort({ a: 1, b: 1 })	{ a: 1, b: 1 }

非最左前缀

例如，data集合创建如下索引：

```
db.data.createIndex( { a:1, b: 1, c: 1, d: 1 } )
```

索引可以支持对索引关键字的非前缀子集进行排序操作。为此，查询必须在排序键之前的所有前缀键中包含相等条件。

下表所示的操作可以使用索引来获取排序顺序。

Example	Index Prefix
db.data.find({ a: 5 }).sort({ b: 1, c: 1 })	{ a: 1 , b: 1, c: 1 }
db.data.find({ b: 3, a: 4 }).sort({ c: 1 })	{ a: 1, b: 1, c: 1 }
db.data.find({ a: 5, b: { $lt: 3} }).sort({ b: 1 })	{ a: 1, b: 1 }

只有排序子集之前的索引字段在查询文档中必须具有相等条件，其他索引字段可以指定其他条件。

如果查询未在排序规范之前或与排序规范重叠的索引前缀中指定相等条件，则操作将无法有效使用该索引。例如，以下操作指定一个{c：1}的排序文档，但查询文档在前面的索引字段a和b中不包含相等匹配项：

```
db.data.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
```

```
db.data.find( { c: 5 } ).sort( { c: 1 } )
```

这些操作不会有效地使用索引{a：1，b：1，c：1，d：1}，甚至可能不会使用索引来检索文档。

4.10 查询计划

慢查询：

```
use helei
db.setProfilingLevel(2)
```

- 关闭；
- 记录超过 50ms 的 db.setProfilingLevel(1,50) ；
- 记录每个读写操作。

```
db.getProfilingStatus()

use local
db.system.profile.find({millis:{$gt:150}}).pretty()查找超过150ms的慢查询
db.system.profile.find().sort({$natural:-1}).limit(1).pretty() 最近的结果显示
explain("executionStats")
```

查询和排序都遵循最左前缀原则。explain()必须放在最后：

```
mongostat
db.runCommand(
   {
     explain: { count: "products", query: { quantity: { $gt: 50 } } },
     verbosity: "queryPlanner"
   }
)


db.runCommand(
   {
```

```
        explain: { count: "products", query: { quantity: { $gt: 50 } } },
        verbosity: "executionStats"
    }
)
```

4.11 systemprofile

system.profile.op 这一项主要包含如下几类：

```
insert
query
update
remove
getmore
command
```

这些代表了该慢日志的种类，包括查询、插入、更新、删除及其他。

```
system.profile.ns
```

该项表明该慢日志是哪个库的哪个集合所对应的慢日志。

```
system.profile.query
```

该项详细输出了慢日志的具体语句和行为。

```
system.profile.keysExamined
```

该项表明为了找出最终结果 MongoDB 搜索了多少个 key。

```
system.profile.docsExamined
```

该项表明为了找出最终结果 MongoDB 搜索了多少个文档。

```
system.profile.keyUpdates
```

该项表明有多少个 index key 在该操作中被更改，更改索引键也会有少量的性能消耗，因为数据库不单单要删除旧 Key，还要在 B-Tree 索引中插入新的 Key。

```
system.profile.writeConflicts
```

写冲突发生的数量，例如，“update”一个正在被别的 update 操作的文档。

```
system.profile.numYield
```

为了让别的操作完成而自己让步的次数，即当前操作让步于其他操作的次数一般发生在需要访问的数据尚未被完全读取到内存中时，MongoDB 会优先完成在内存中的操作。

```
system.profile.locks
```

在操作中产生锁的种类有多种，如下表所示。

Global	Represents global lock.
MMAPV1Journal	Represents MMAPv1 storage engine specific lock to synchronize journal writes; for non-MMAPv1 storage engines, the mode forMMAPV1Journal is empty.
Database	Represents database lock.
Collection	Represents collection lock.
Metadata	Represents metadata lock.
oplog	Represents lock on the oplog.

锁的模式也有多种，如下表所示。

Lock Mode	Description
R	Represents Shared (S) lock.
W	Represents Exclusive (X) lock.
r	Represents Intent Shared (IS) lock.
w	Represents Intent Exclusive (IX) lock.

```
system.profile.locks.acquireCoun
```

该操作是指在各种不同的种类下请求锁的次数。

```
system.profile.nreturned
```

该操作最终返回文档的数量。

```
system.profile.responseLength
```

该操作返回结果的大小，单位为 byte，该值如果过大，则需考虑利用 limit()等方式减少输出结果。

```
system.profile.millis
```

该操作返回从开始到结束的耗时，单位为毫秒。

```
system.profile.execStats
```

该操作包含了一些相关的统计信息，只有 query 类型的操作才会显示。

```
system.profile.execStats.stage
```

该操作包含了相关的详细信息，例如，是否用到索引。

```
system.profile.ts
```

该操作是指执行时的时间。

```
system.profile.client
```

该操作是由哪个客户端发起的，并显示该客户端的 IP 地址或 hostname。

```
system.profile.allUsers
```

该操作是由哪个认证用户执行的。

```
system.profile.user
```

该操作是否由认证用户执行，如果认证后使用其他用户操作，则该项为空。

4.12　Profile 操作相关

```
db.getProfilingStatus()
```

db.setProfilingLevel(1,500)　#开启慢查，记录 500ms 以上的操作。0 为不记录，1 为记录慢查默认为 100ms，2 为记录全部操作

db.runCommand({ profile: 2, slowms: 200, ratelimit: 100 })#percona 版本的 MongoDB 具备的

ratelimit（范围 1～1000），ratelimit 只会限制快查询的频率，本条命令实现每 100 条快查记录 1 条，慢查询超过 200ms 的全部记录。配置为 0 和 1 是一样的效果，记录所有快查

```
    db.system.profile.find().limit(10).sort({ts:-1}).pretty()#返回最近的10条操
作
    db.system.profile.find().limit(10).sort({millis:-1}).pretty()#返回最耗时的
10条操作
    db.system.profile.find({op:{$ne:'command'}}).pretty()#除了 command 类型的操
作
    db.system.profile.find({ns:"your_db.mycollection"}).pretty() #返回特定集合
    db.system.profile.find({millis:{$gt:150}}).pretty()#查找超过150ms的慢查询
    db.system.profile.find(
    {
    ts : {
    $gt : new ISODate("2017-03-22T03:00:00Z") ,
    $lt : new ISODate("2017-08-16T13:40:00Z")
    }
    }
    ).pretty()


    db.system.profile.find(
    {
    ts : {
    $gt : new ISODate("2017-03-22T03:00:00Z") ,
    $lt : new ISODate("2017-08-16T13:40:00Z")
    }
    },
    { user : 0 }
    ).sort( { millis : -1 } ).pretty()
```

第 5 章 备份与恢复

随着网络时代的飞速发展，各大公司和企业对于信息安全的重要性也越来越重视。但是数据备份的重要性可能经常被忽视。这就好比我们的身体正处于一个亚健康的状态，自我感觉良好，但不知危险正在靠近。没有数据库的备份就没有恢复，当服务器意外宕机或者误操作造成大量数据丢失时，对于数据安全性要求很高的行业（金融类、电商类、游戏类）来说，就会造成重大的经济损失。所以提高数据库的高可用性和遇到灾难的可修复性就显得至关重要了。本章介绍 MongoDB 的备份策略和方法。

5.1 逻辑备份

```
    mongodump -h 127.0.0.1:28000 -u sys_admin --authenticationDatabase admin -p
MANAGER    -d    databasename-c    collationname    --dumpDbUsersAndRoles    -o
/home/work/helei.sql
```

-h：MongoDB 所在服务器地址，例如，127.0.0.1。当然也可以指定端口号：127.0.0.1:27017。

--port：端口号。

-d：需要备份的数据库实例，例如，test。

-c：需要备份的集合。

-o：备份数据的存放位置。

-u：用户名。

-p：密码。

--gzip：压缩。

--oplog：point in time 恢复用，只支持全库备份。

--authenticationDatabase：认证库。

--dumpDbUsersAndRoles

dump 用户和角色，只有在单库备份时才需要这么做。

--archive=dbname.gz：3.2 版本新增，不能和-o 同时使用。

归档备份为 1 个文件，但不能和-o 同时使用。

5.2 Oplog Replay

目标

由于生产环境人为的误操作，导致将整个文档错误更新，目标是利用 oplog 实现 point-in-time 的恢复，即恢复到误操作前的状态。

准备

副本集环境 3 台，一主两从，Standalone 实例一台。

恢复步骤

mongodump 备份出 oplog：

```
mongodump -h 127.0.0.1:27020 -u sys_admin --authenticationDatabase admin -p
MANAGER -d local -c oplog.rs -o /home/work/backup
```

bsondump 让 oplog 具备可读性：

```
bsondump /home/work/backup/local/oplog.rs.bson >/home/work/local.log
```

将 oplog 改名：

将 oplog 文件改名，避免在 restore 时报错，并将 oplog 的备份复制到 Standalone 机。

```
mv oplog.rs.bson oplog.bson
```

Standalone 实例导入物理或逻辑备份。

Standalone 库启动。

导入全库备份。

找出误操时的具体时间。

预计在 2017-08-21 16:55:47 时删除，转换为 1503305747。

这里如果不能精确到秒，则可以在 16:55 左右时删除。

```
cat local.log |grep -A 100 150330570, 发现是在 1503305747
```

执行恢复操作

找到误操作的时间点，会截止在这个时间点不恢复：

```
mongorestore -h 127.0.0.1:27030 --oplogReplay --oplogLimit 1503305747:1
/home/work/backup/
```

恢复到生产环境

人工恢复到生产环境一般需要停业务，因此最好是将备份导出到一个独立集群供研发读取，有应用进行导入和修正。

5.3　物理备份

物理备份非常简单，在从库上执行如下命令就可以完成物理备份：

```
heleitest:PRIMARY> use admin
switched to db admin
heleitest:PRIMARY>     db.runCommand({createBackup:     1,     backupDir:
"/home/work/tmp"})
```

将备份文件复制到待恢复机器上，使用 mongod 直接拉起 Mongo 进程就可以完成物理恢复。

第 6 章 高可用架构集群管理

在 ACMUG 大会上，我演讲过一个主题，MySQL 和 MongoDB 绝非是替代的关系，各自都有其特点。不同的数据库只是应用的场景不同而已。MongoDB 核心的两个特点，第一是副本集的自动切换，保证数据的高可靠、服务的高可用性；第二是自动分片、服务的横向扩展能力。第 6 章的主要内容就是围绕这两大特性，深入分析 MongoDB 的特性，以便于可以更好地服务于公司现有的业务发展。

6.1 副本集

MongoDB 中的副本集是一组保持相同数据集的 mongod 进程。副本集提供冗余和高可用性是所有生产部署的基础。本节介绍 MongoDB 中的复制及副本集的组件和体系结构。

6.1.1 冗余和数据可用性

复制提供冗余并增加数据可用性。在不同数据库服务器上有多个数据副本的情况下，复制可以提供一定的容错能力，以防止单个数据库服务器的丢失。

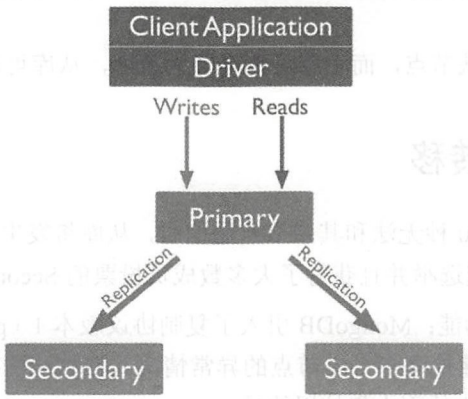
在某些情况下，复制可以提高读取容量，因为客户端可以将读取操作发送到不同的服务器。在不同的数据中心维护数据副本可以增加分布式应用程序的数据本地化和可用性。还可以为了专门目的而保留其他副本，例如，灾难恢复、报告或备份。

6.1.2　MongoDB 中的副本集

副本集是一组保持相同数据集的 mongod 实例，它包含多个数据承载节点和一个可选的仲裁节点。在数据承载节点中，只有一个成员是主节点，而其他节点是从节点。

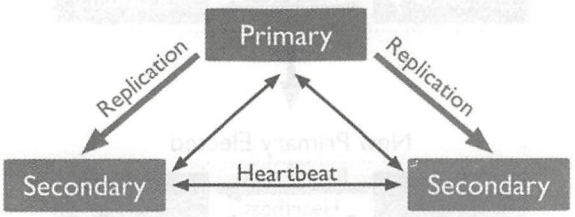
主节点接收所有的写入操作，一个副本集只能有一个主节点响应应用的{ w: "majority" } write concern。尽管在某些情况下，另一个 mongod 实例可能暂时认为自己也是主节点的一部分。主节点会通过 oplog 记录所有在主节点让数据发生改变的操作。

下图显示了一个基本的 3 节点副本集架构。



Secondary 节点会复制主节点的 oplog 日志信息，并在自己的节点上应用这些 oplog，类似 MySQL 的从库。如果主库出现无法访问的情况，则从库会发起投票，选举出一个新的 Primary 节点。

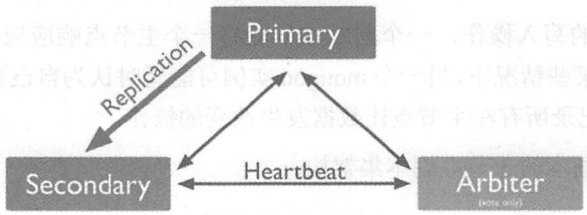
下图显示副本集之间的复制模式和心跳检测。



可以将一个 mongod 实例作为仲裁节点添加到副本集中。仲裁节点不维护数据集，它的目的是响应其他副本成员的心跳和选举请求。因为不存储数据集，所以仲裁节点可以成为提供副本集仲裁功能的好方法，它并不需要太好的硬件资源支撑。因此，在新版本中不再推荐使用仲

裁节点。

下图是预算不够的情况下使用仲裁节点代替 Secondary 节点的架构图，生产环境中不建议使用仲裁节点。



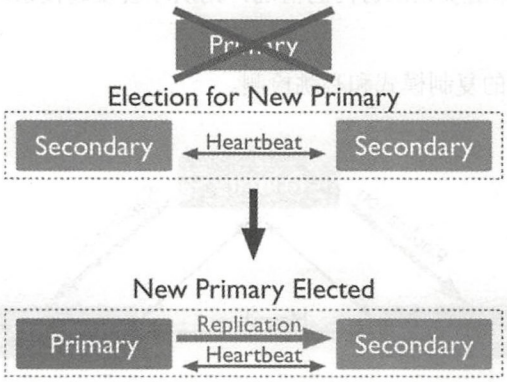
仲裁节点将永远是仲裁节点，而主库可能会降为从库，从库可以通过投票变为新的主库。

6.1.3 自动故障转移

当 Primary 节点连续 10 秒无法和其余节点通信时，从库将发生选举，在从库中选出一个库作为新的主库。第一个发起选举并且获得了大多数成员投票的 Secondary 节点，将成为主节点。

在 3.2 版本中的新增功能：MongoDB 引入了复制协议版本 1（protocolVersion：1），以减少副本集故障转移时间并加速检测多个主节点的异常情况。在默认情况下，新版本的副本集将使用 protocolVersion1，老版本的副本集使用的是 protocolVersion0。

下图为选举过程示意图。



故障切换过程通常会在一分钟内完成。副本集的成员可能需要 10 到 30 秒才能声明 Primary 不可访问（相关参数为--electionTimeoutMillis）。

从 MongoDB 3.2 开始加强了复制选举功能，MongoDB 减少了副本集故障转移的时间。

6.1.4　关于 MongoDB 的读操作

在默认情况下，客户端从 Primary 读取数据，但是客户端可以指定读取偏好，以将读取操作发送给从库。复制是异步的，所以从库返回的数据可能不是最新的。

在 MongoDB 中，客户端可以在写入真正落盘之前看到写入的结果，即 Read Uncommitted。

无论 write concern 是如何配置的，当客户端使用“local”或“available” readConcern 时，其他客户端都可以在写入操作发给客户端确认之前看到写入操作的结果。

使用“local”或“available” readConcern 的客户端可以读取可能随后回滚的数据。

6.2　副本集成员状态

成员与状态描述如下表所示。

数　字	名　称	状 态 描 述
0	STARTUP	还不是任何集合的活动成员。所有的成员在该状态启动。在 StartUp 状态，mongod 解析副本集配置文档
1	PRIMARY	处于 Primary 状态的成员是唯一能接收写操作的成员
2	SECONDARY	处于 Secondary 状态的成员复制数据存储
3	RECOVERING	可以参与选举。成员要么在实施启动自检测，要么完成回滚或重新同步的转换
5	STARTUP2	成员加入了集合，正运行初始化同步
6	UNKNOWN	成员的状态，正如从集合的另一个成员中所看到的，未知
7	ARBITER	仲裁不复制数据，仅参与选举
8	DOWN	该成员从其他成员看起来，不可达
9	ROLLBACK	该成员正在实施回滚。数据不可读
10	REMOVED	成员曾经在副本集但随后被移除

6.3　副本集原理

Secondary 初次同步数据时，会先进行 init sync 操作，从 Primary（或其他数据更新的 Secondary）同步全量数据，然后不断通过 tailable cursor 从 Primary 的 local.oplog.rs 集合里查询最新的 oplog 并应用到自身。

initial sync 的过程：

步骤 1：T1 时间，从 Primary 同步所有数据库的数据（local 除外），通过 listDatabases + listCollections + cloneCollection 命令组合完成，假设 T2 时间完成所有操作。

步骤 2：从 Primary 应用[T1-T2]时间段内的所有 oplog，可能部分操作已经包含在步骤 1 中，但由于 oplog 的幂等性，可重复应用。

步骤 3：根据 Primary 各集合的 index 设置，在 Secondary 上为相应集合创建 index（每个集合_id 的 index 已在步骤 1 中完成）。

假设复制集内投票成员（后续介绍）的数量为 N，则大多数为 $N/2+1$，当复制集内存活成员数量不足大多数时，整个复制集将无法选举出 Primary，复制集将无法提供写服务，处于只读状态。

之所以使用奇数个节点，是由于偶数节点和奇数节点容忍失效数一致，所以基数节点从成本和冗余角度讲更具性价比。

下表简单显示了集群中投票成员数和对应的“大多数”节点阈值，以及对应的集群容忍可失效的节点数。

投票成员数	大 多 数	容忍失效数
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

6.4 复制集成员

MongoDB 中的副本集是一组提供冗余和高可用性的 mongod 进程。副本集的成员是 Primary。

Primary 节点要接收所有写入操作。

Secondary 节点会从 Primary 节点复制操作，它们之间的数据是相同的。Secondary 节点可以配置额外的配置选项用以特殊作用。例如，可以为 Secondary 节点配置 Vote 0 或者 Pirority 0 选项。

也可以在副本集中建立一个仲裁节点，仲裁节点不会像其他 Secondary 成员一样保留数据。但是，如果当前的 Primary 节点不可用，则其能够作为投票节点对新的主节点选举进行投票。

副本集架构最低建议配置是 3 个成员，1 个 Primary 节点和 2 个 Secondary 节点。也可以部署 1 个一主一从一仲裁节点的架构，不过从冗余和高性能上来说，推荐使用一主两从节点，而不使用仲裁节点。

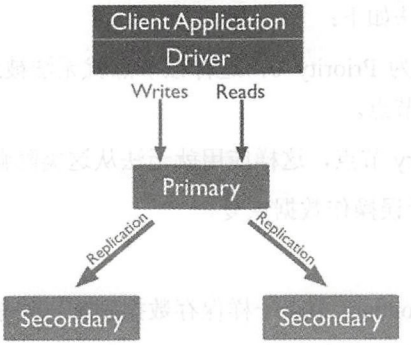
MongoDB 从 3.0 版本起，副本集可以有多达 50 名成员，但只有 7 个有投票权的成员。在以前的版本中，副本集最多可以有 12 个成员。

Primary

Primary 节点是副本集中接收写入操作的唯一成员。MongoDB 在 Primary 服务器上应用写操作，然后在 Primary 节点上使用 oplog 记录操作。Secondary 成员复制 oplog 并将操作应用于 Secondary 上。

在以下三成员副本集中，Primary 要接收所有写入操作，然后 Secondary 复制 oplog 并应用到它们的数据集中。

下图显示了一个最简单的副本集架构，以及客户读写主库的情况。



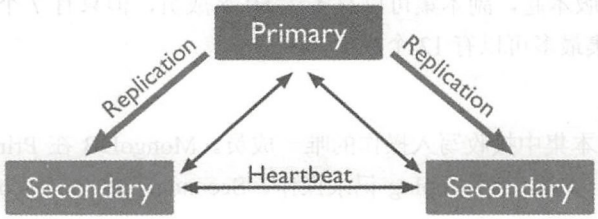
副本集的所有成员都可以接收读取操作。但在默认情况下，应用程序将其读取操作指向 Primary 节点。

副本集架构中有且只能有一个 Primary 节点。如果当前 Primary 节点不可用，则会选举确定新的 Primary 节点。

Secondaries

Secondary 上保存的是 Primary 节点的数据副本。Secondary 节点将 Primary 节点的 oplog 日志中的操作异步应用到自己的节点上。副本集可以有一个或多个 Secondary 节点。

下图的架构中包含两个 Secondary 成员。Secondary 节点复制 Primary 节点的 oplog 日志并应用在自己的节点上。



虽然客户端不能将数据写入 Secondary 节点，但客户端可以从 Secondary 节点读取数据。

Secondary 节点可以被选举为新的 Primary 节点。如果当前的 Primary 节点不可用，则副本集将发起选举，并投票选举出一个新的 Primary 节点。

Secondary 节点配置的方法如下：

可以配置 Seconday 节点为 Priority 0，这样该节点就无法被选举为 Primary 节点，这样的 Secondary 节点可以用作冷备节点。

可以配置隐藏的 Secondary 节点，这样应用就无法从这类隐藏的节点上读取数据。

可以配置延迟节点，用于误操作数据恢复。

Arbiter

仲裁节点不会像其他 Secondary 节点一样保存数据副本，仲裁节点不会被选举为主节点。

从 MongoDB 3.6 起，仲裁节点的 Priority 会默认配置为 0。

6.5 复制集成员类型

Arbiter

Arbiter 节点只参与投票，不能被选为 Primary，并且不从 Primary 同步数据。

比如部署了一个两个节点的复制集、1 个 Primary、1 个 Secondary，如果任意节点宕机，则复制集将不能提供服务了（无法选出 Primary），这时可以给复制集添加一个 Arbiter 节点，此时即使有节点宕机仍能选出 Primary。

Arbiter 本身不存储数据，是非常轻量级的服务，当复制集成员为偶数时，最好加入一个 Arbiter 节点，以提升复制集的可用性。

从 3.6 版本起，Arbiter 必须设置 Priority 为 0。

Priority 0

Priority0 节点的选举优先级为 0，不会被选举为 Primary。

比如跨机房 A、机房 B 部署了一个复制集，并且想指定 Primary 必须在 A 机房，这时可以将 B 机房的复制集成员 Priority 设置为 0，这样 Primary 就一定会是 A 机房的成员（注意：如果这样部署，则最好将“大多数”节点部署在 A 机房，否则网络分区时可能无法选出 Primary）。

从 3.2 版本起，Priority 大于 0 的 vote 必须为 1。

从 3.2 版本起，vote=0 的 Priority 必须为 0。

如果低优先级的 Secondary 被高优先级的 Secondary 选举为 Primary，则它会继续选举直到高优先级的 Secondary 成为 Primary。

有类似的日志：

```
2017-11-06T13:50:33.252+0800 I REPL     [ReplicationExecutor] not electing
self, primary:37018 would veto with 'secondary:37018 has lower priority than
primary:37018'
```

Vote 0

MongoDB 3.0 的复制集成员最多 50 个，参与 Primary 选举投票的成员最多为 7 个，其他成员（Vote0）的 vote 属性必须设置为 0，即不参与投票。

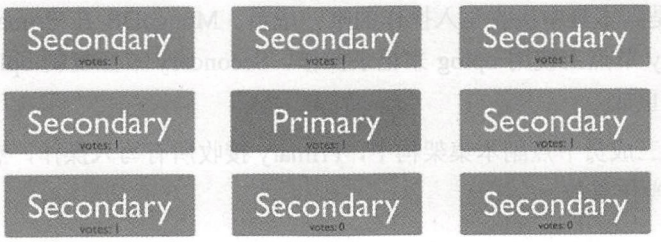
从 3.2 版本起，vote=0 的必须 Priority 为 0，也就是不能成为主节点。

从 3.2 版本起，vote=1 的只能从 vote=1 的节点同步。

只有这几个状态的节点才能发起选举：

```
PRIMARY
SECONDARY
RECOVERING
ARBITER
ROLLBACK
```

下图显示了一个 9 节点架构中，最多的具备 vote 权限节点数至多只能为 7 个，剩下的 Secondary 节点必须为非投票节点即 vote=0。



Hidden

Hidden 节点不能被选为主（Priority 为 0）节点，并且对 Driver 不可见。

因 Hidden 节点不会接收 Driver 的请求，可使用 Hidden 节点做一些数据备份、离线计算的任务，或者不会影响复制集的服务。

Delayed

Delayed 节点必须是 Hidden 节点，并且其数据落后于 Primary 一段时间（可配置，比如 1 个小时）。

因 Delayed 节点的数据比 Primary 落后一段时间，当错误或者无效数据写入 Primary 时，可通过 Delayed 节点的数据来恢复到之前的时间点。

副本集 rollback 过程

回滚只会发生在主节点的写操作没能成功再从节点上应用就“Down”掉的情况下。当主节点重新以一个从节点加入复制集时，它将进行“回滚”，其上的写操作将与复制集中其他成员的写操作保持一致。

rollback 的数据不能超过 300MB。

如何避免 rollback？w 默认为 1，使用 majority 来保证写操作已经达到大多数从节点：

```
{ w: <value>, j: <boolean>, wtimeout: <number> }
```

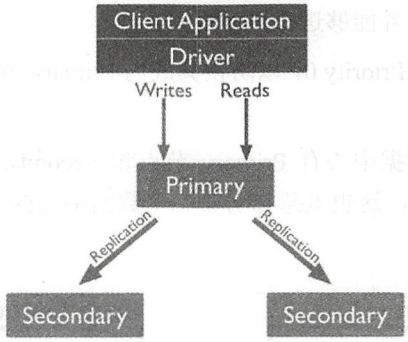
当 Primary 宕机时，如果有数据未同步到 Secondary，当 Primary 重新加入时，如果新的 Primary 上已经发生了写操作，则旧 Primary 需要回滚部分操作，以保证数据集与新的 Primary 一致。

旧 Primary 将回滚的数据写到单独的 rollback 目录下，数据库管理员可根据需要使用 mongorestore 进行恢复。

6.6 副本集中的主库

Primary 节点是副本集中接收写入操作的唯一成员。MongoDB 在 Primary 节点上应用写操作，然后在 Primary 节点上使用 oplog 来记录操作。Secondary 节点复制 oplog 日志，并将操作在自己节点上进行重放。

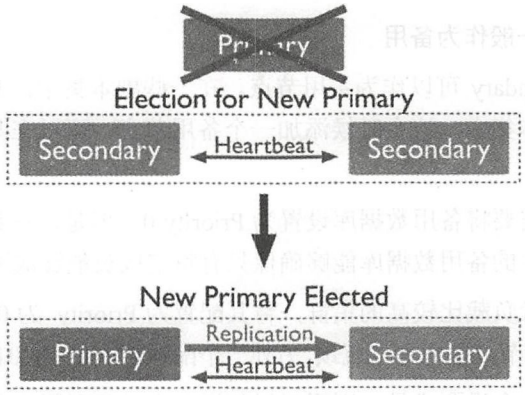
在下图所示的三成员节点副本集架构中，Primary 接收所有写入操作，然后 Secondary 节点复制 oplog 来进行操作重放。



副本集的所有成员都可以接收读取操作。但在默认情况下，应用程序将其读取操作指向 Primary。

副本集可以有至多一个 Primary 节点，Primary 节点宕机后，集群会触发选举以选出新的 Primary 节点。

在下图所示的三成员节点副本集架构中，Primary 宕机后触发了一次选举，从剩下的两个 Secondary 节点里选举出了一个新的 Primary 节点。



6.7　副本集中的从库

6.7.1　Priority 0 从库

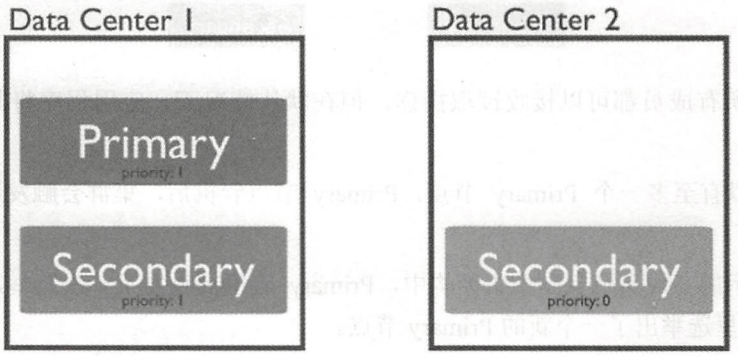
Priority 0 的成员不能成为 Primary 的成员，不能主动触发选举。

除了上述限制，具有 Priority 0 的 Secondary 节点与正常的 Secondary 无异：它们一样持有

数据的副本，接收读取操作，并能够进行选举投票。

将 Secondary 节点配置为 Priority 0，以防止其成为 Primary 节点，这在多数据中心部署中特别有用。

例如，在下图中，一个数据中心有 Primary 节点和 Secondary 节点，第二个数据中心拥有 Priority 0 的 Secondary 节点，这也就意味着第二个数据中心的 Secondary 节点永远不能成为 Primary 节点。



优先级为 0 的成员一般作为备用

优先级为 0 的 Secondary 可以作为备用节点。在一些副本集中，某些情况下可能不能直接添加一个新的成员到副本集中，这个时候添加一个备用节点，可以在某个成员不可用时用于替换成员操作。

在许多情况下，不需要将备用数据库设置为 Priority 0。但是，在具有不同硬件或地理分布的副本集中，优先级为 0 的备用数据库能够确保只有特定成员能够成为主数据库。

对于一些硬件差或者负载比较高的集群，将其配置为 Priority 为 0，让其不能成为主节点。避免性能差的机器成为主节点造成集群瓶颈。另外一个配置 Priority 为 0 的场景是配置隐藏节点。

如果副本集已经有 7 个投票成员，则要记得新增节点时要配置为非投票节点。

故障转移注意事项

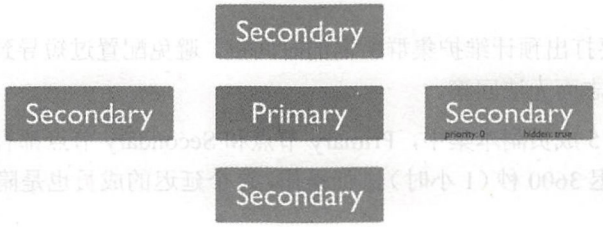
将 Secondary 节点配置为 Priority 0 时，要考虑潜在的故障切换模式，例如，所有可能的网络分区。始终确保主数据中心有有效的投票节点，以及有资格成为 Primary 节点的成员。

6.7.2 hidden 从库

隐藏节点也拥有数据的副本，但对客户端应用程序不可见。隐藏节点的工作负载比正常节

点要低很多，因此可以作为其他用途，例如，备份。隐藏节点必须始终是 Priority 0 的成员，因此不能成为 Primary 节点。db.isMaster()命令不显示隐藏的成员，隐藏的成员也会参加选举。

在下图所示的 5 节点的副本集中，所有 Secondary 节点都具备数据的副本，但其中一个 Secondary 节点是隐藏的。



客户端不会将只读流量发给隐藏成员。因此，除了基本复制，这些成员不会收到任何流量。我们可以使用隐藏的成员执行专门的任务，例如，报告和备份。延迟节点应该配置为隐藏，隐藏的成员可以在复制集选举中投票。所以在停止这类隐藏节点的同时，也应当确认集群中具备投票节点的成员是否还具备大多数成员的条件，否则主库会被降级为从库，整个集群变为只读状态。

如果使用 MMAPv1 存储引擎，则可以使用 db.fsyncLock()和 db.fsyncUnlock()命令来让所有正在进行的写操作落盘并锁定整个实例来阻止写入，以便在备份操作期间刷新所有写入并锁定 mongod 实例。

从 MongoDB 3.2 起，db.fsyncLock()可以确保使用 MMAPv1 或 WiredTiger 存储引擎的 MongoDB 实例的数据文件不会更改，从而为创建备份提供一致性保障。

在之前的 MongoDB 版本中，db.fsyncLock()不能保证 WiredTiger 引擎在做 CP 复制时的一致性。

6.7.3　延迟从库

延迟节点也拥有副本集的副本，然而延迟成员的数据集反映了该集合的延迟状态。例如，当前时间是 09:52，并且成员有一个小时的延迟，则延迟节点没有比 08:52 更往后的操作。

由于延迟成员是正在运行集群的“历史”快照，因此可以帮助你从各种人为错误中恢复过来。例如，延迟的成员可以从误操作中恢复数据。

前提：

（1）必须是 Priority=0，配置 Priority=0 来防止延迟节点成为 Primary 节点。

（2）配置 hidden=true，避免应用从延迟节点上读到过期数据。

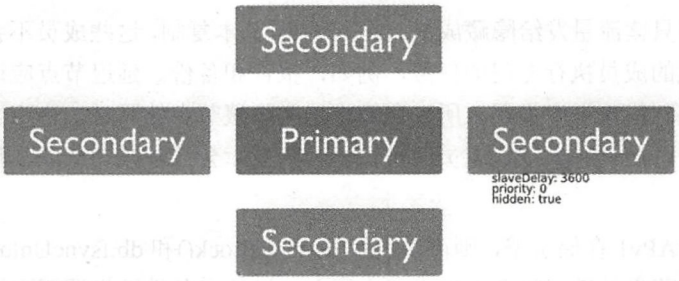
（3）如果 votes=1，那么延迟节点也具备投票权限。

延迟节点延迟复制和应用 Primary 节点的 oplog，在选择延迟量的时候应当考虑：

（1）延迟时间要低于 oplog Window，以避免超过 oplog Window 时主库已经删除了老的 oplog 日志。

（2）延迟时间要打出预计维护集群所需的时间量，避免配置过短导致故障出现时，延迟节点已经应用了错误日志而无法回滚。

在如下图所示的 5 成员副本集中，Primary 节点和 Secondary 节点都有数据的副本。其中一名 Secondary 成员延迟 3600 秒（1 小时）进行操作，这个延迟的成员也是隐藏的，并且是 Priority 0 的成员。



6.8 oplog 简介

oplog（操作日志）是一个特殊的固定集合，它保存了数据库中所有的数据变化记录，MongoDB 在 Primary 上应用数据库操作，然后在 Primary 上使用 oplog 记录操作，Secondary 节点异步复制并应用这些操作。所有副本集成员都在 local.oplog.rs 集合中包含一个 oplog 的副本。

为了便于复制，所有副本集成员都将心跳（ping）发送给所有其他成员，每一个成员都可以从任何其他成员导入和应用 oplog。

oplog 中的每个操作都是幂等的。也就是说，不论在库中应用一次或者多次 oplog，产生的结果都不会变。

oplog 的大小如下表所示。

Storage Engine	Default Oplog Size	Lower Bound	Upper Bound
In-Memory Storage Engine	5% of physical memory	50 MB	50 GB
WiredTiger Storage Engine	5% of free disk space	990 MB	50 GB
MMAPv1 Storage Engine	5% of free disk space	990 MB	50 GB

从这张表我们能够看出，在默认的情况下，oplog 最大磁盘剩余空间的 5%，即 50GB，但 oplog 最大不是 50GB，可以通过命令行手动进行指定。

在 mongod 创建 oplog 之前，可以使用 oplogSizeMB 选项指定其大小。一旦首次启动副本集成员，可使用 replSetResizeOplog 管理命令更改 oplog 大小。replSetResizeOplog 使我们可以动态调整 oplog 的大小，无须重新启动 mongod 进程（replSetResizeOplog 是 MongoDB 3.6 的新特性，老版本还需要停库来重新设定 oplog 大小）。

6.9　oplog 过滤

查询 oplog 时间大于 xx 的对 ccc.BaseProfiles 集合的删除操作：

```
db.oplog.rs.find({ns:"ccc.BaseProfiles",op:"d","ts":{$gte:Timestamp(1502
640000,0)}}).count()
```

- ts：操作发生的时间；
- h：记录的唯一 ID；
- v：版本信息；
- op：写操作的类型；
- n：no-op；
- c：db cmd；
- i：insert；
- u：update；
- d：delete；
- ns：操作的 namespace，即数据库、集合；
- o：操作所对应的文档；
- o2：更新时所对应的 where 条件，更新时才有。

6.10　副本集的数据复制

为了保持自身的数据是最新的，副本集的次要成员同步或复制来自其他成员的数据。MongoDB 使用两种形式的数据同步：新成员的初始同步，以及已有成员之间的持续复制。

初始化同步

初始同步将副本集的一个成员的所有数据复制到另一个成员。

初始化同步的过程：

会复制除 local 库外的所有数据库。在复制期间，mongod 扫描每个源数据库中的每个集合，并把它们进行复制。

在 MongoDB 3.4 中，初始同步在为每个集合复制文档时构建所有集合索引。在早期版本的 MongoDB 中，这个阶段只有_id 索引被建立，其他的索引则是在数据复制完成后重建。

在 MongoDB 3.4 中，初始同步会在数据复制期间拉取新添加的 oplog 记录，以确保目标成员的 local 数据库中有足够的磁盘空间，并在此数据复制阶段期间临时存储这些 oplog 记录。

根据源数据库的 oplog，将所有更改应用于数据集，mongod 在应用这些数据时会变更自身的状态。初始同步完成后，成员状态从 StartUp2 转换到 Secondary。

初始化同步中改集合名

从 3.2.12 版本起，如果初始同步正在运行时同步源上的集合已重命名，则目标成员的初始同步将失败并重新启动以避免数据损坏。

复制

Secondary 成员在初始同步后不断复制数据，即将 oplog 从其源同步中复制，并在异步过程中应用这些操作。

Secondary 可以根据 ping 时间和其他成员复制状态的变化自动选择同步源。

从 MongoDB 3.2 起，具有 vote 权限的 MongoDB 成员不能从不具备 vote 权限的成员同步。

Secondary 会避免从隐藏节点和延迟节点同步。

如果 Secondary 成员将 members[n] .buildIndexes 设置为 true，则只能从 buildIndexes 为 true 的其他成员进行同步。buildIndexes 为 false 的成员可以从任何其他成员同步，buildIndexes 默认是 true。

多线程复制

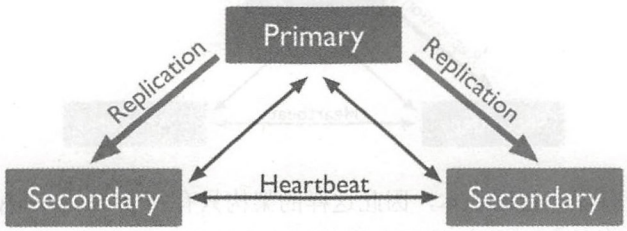
MongoDB 使用多线程来进行批量写入操作以提高并发性。MMAPv1 引擎以 namespace（MMAPv1 引擎）为 groups，WriedTiger 引擎以文档 ID（WiredTiger 引擎）为 groups，将写入操作“groups batches”化，同时使用不同的线程来应用每组操作。MongoDB 始终将写入操作以原始写入顺序应用于给定文档中。

在应用批处理时，MongoDB 将阻止所有读取操作。因此，Secondary 的读取查询永远不会返回在 Primary 上还没有写入的数据。

6.11　3 节点最小副本集架构

副本集的最小架构要有三个成员。一个三成员副本集可以有三个持有数据的成员，或两个持有数据的成员和一个仲裁节点。

Primary + 2Secondary 架构如下图所示。

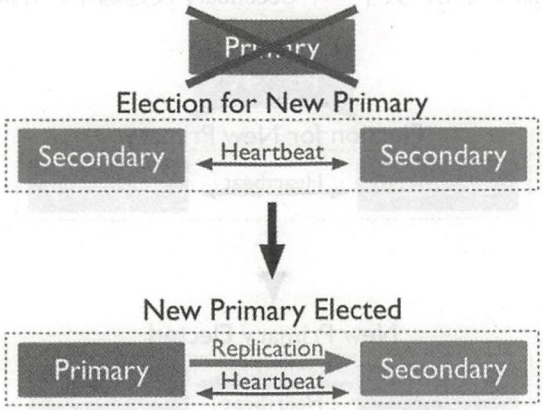


具有 3 个存储数据成员的副本具有：

- 1 个 Primary。
- 两名 Secondary 成员，两个 Secondary 节点都可以在选举中成为 Primary 节点。

除 Primary 节点外，两个 Secondary 节点也具备完整的数据副本，副本集架构能够为集群带来容错能力和高可用性。如果 Primary 服务器不可用，则副本集会选择其中一个 Secondary 成员作为 Primary 服务器并继续正常操作，而旧的 Primary 服务器在可用时会重新加入副本集。

下图为 3 节点副本集成员主库宕机后新主库被选举和从库复制的示意图。

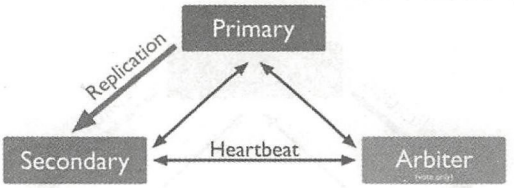


Primary+Secondary+Arbiter 架构

Primary 节点和 Secondary 节点具有数据的副本，这样的 3 成员副本集具有：

- 一个 Primary 节点。
- 一个 Secondary 成员，Secondary 节点可以被选举为新的 Primary 节点。
- 一名仲裁员。仲裁员只在选举中投票，而不具备数据的副本。

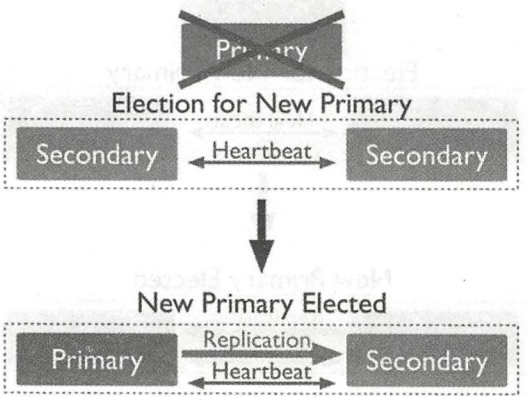
下图为 3 节点副本集成员，由仲裁节点组成参与的最简架构。



由于仲裁节点没有保存数据副本，因此这样的架构只有一个 Secondary 具备数据副本。但是仲裁节点需要更少的资源，代价是更有限的冗余和容错。推荐在生产库中尽量不使用仲裁节点。

6.12 副本集的选举

副本集使用选举机制来确定哪个成员将成为 Primary 成员。在启动副本集之后进行选举，以及任何时候 Primary 节点不可用时，也会发生选举。Primary 节点是副本集架构中唯一可以接收写入操作的成员。如果一个 Primary 节点不可用，则副本集架构会选举出一个新的 Primary 节点，使整个集群恢复写入功能，这并不需要人工的介入，是自动化完成的。在下图所示的 3 节点副本集中，Primary 节点不可用，其中一个 Secondary 发起选举，并成为新的 Primary 节点。



选举对副本集架构来讲非常重要，在发生选举时需要一定的时间才能完成。在进行选举时，副本集没有 Primary 节点，并且不能接收写入操作，所有其余成员都变为只读状态。

如果副本集的大部分成员不可访问或不可用，则 Primary 节点将降级为 Secondary 节点。在发生这种情况后，副本集不能接收写入，如果 read concern 配置的不是 Primary，那么集群中的 Secondary 节点依然可以提供只读功能。

影响选举的因素

从 MongoDB 3.2 起，MongoDB 引入了复制协议版本 1（protocolVersion1），以减少副本集故障转移时间，并加速检测多个同时发生的主节点。在默认情况下，新的副本集将使用 protocolVersion1，以前版本的 MongoDB 使用 protocolVersion0。

心跳检测

副本集成员每两秒发送一次心跳（ping），如果心跳在 10 秒内没有返回，则其他成员将没有返回心跳信息的成员标记为不可访问。

节点的优先级

在副本集架构具有稳定的 Primary 节点之后，选举算法将会尽力去让有最高优先级的 Secondary 发起选举。优先级 Priority 的配置会影响选举的时间和最终的选举结果。具有较高 Priority 的 Secondary 相比较低 Priority 的 Secondary 会更早地发起选举，也更容易成为新的 Primary 节点。但是即使优先级较低的 Secondary 也可以在短时间内被选为 Primary，这种情况发生的时候副本集会继续进行选举，直到可用的最高优先级 Secondary 被选举为 Primary 节点。

Priority 为 0 的成员不能成为 Primary 节点，也不会主动发起选举。

投票

副本集成员配置 members[n] .votes 及成员状态决定了成员是否在选举中投票。

从 MongoDB 3.2 起：

（1）无投票权的成员必须配置 Priority 为 0。

（2）Priority 大于 0 的成员其 vote 必须配置为 1。

只有以下几种状态的成员有投票权：

```
PRIMARY
SECONDARY
RECOVERING
ARBITER
ROLLBACK
```

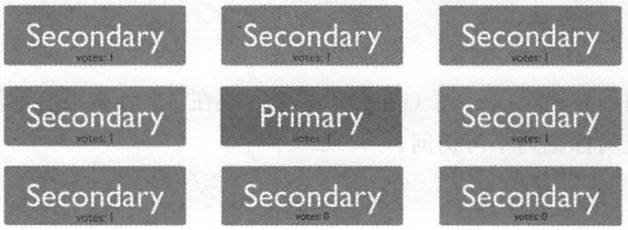
非投票节点

虽然没有投票权的成员在选举中不能投票，但这些成员持有副本集的数据副本，并且可以接收来自客户端应用程序的读取操作。

因为副本集最多可以有 50 个成员，但最多只能有 7 个有投票权的成员。

没有投票权的成员必须配置 Priority 为 0，也就是不能成为 Primary 节点。

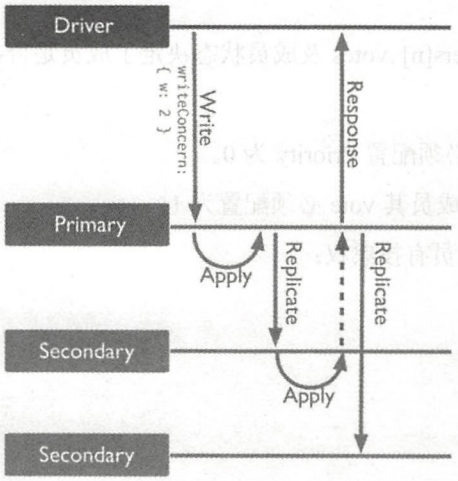
例如，下图所示的 9 成员副本集有 7 个节点有投票权，两个节点没有投票权。



6.12.1 writeConcern

在默认情况下，Primary 完成写操作即返回，Driver 可通过设置 writeConcern 来设置写成功的规则。

下图为当 writeConcern 配置为 w:2 时，driver 写入副本集时的示意图。



要覆盖默认的 writeConcern，需要在每个写入操作的后面指定 writeConcern。例如，以下方法包含 writeConcern，其指定了写入在主节点完成，并且传入至少一个 Secondary 节点，5s 后超时：

```
db.products.insert(
   { item: "envelopes", qty : 100, type: "Clasp" },
   { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

配置 timeout 的方式，可以避免写入无限等待下去，阻塞写操作。例如，一个集群中目前只有 3 个节点可用，但配置了 w:4，要求 4 个节点都写入成功，这时这个写入就会被阻塞。配置了 wtimeout 可以让这个写入命令在达到阈值后返回错误，不会一直被阻塞。

上面的设置方式是针对单个请求的，也可以修改副本集默认的 writeConcern，这样就不用每个请求单独设置：

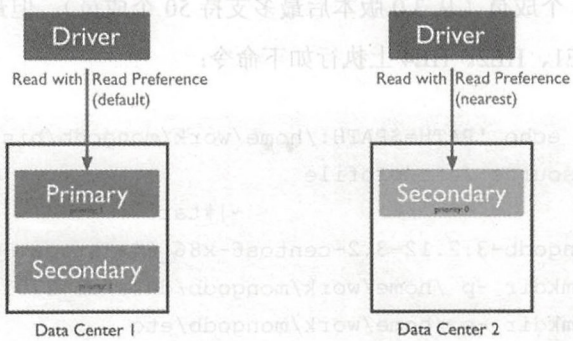
```
cfg = rs.conf()
cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
rs.reconfig(cfg)
```

只有在更大多数投票节点间保持网络连通，才有机会被选 Primary；如果 Primary 与大多数的节点断开连接，Primary 会主动降级为 Secondary。当发生网络分区时，可能在短时间内出现多个 Primary，故 Driver 在写入时，最好设置“大多数成功”的策略，这样即使出现多个 Primary，也只有一个 Primary 能成功写入大多数。

6.12.2　Read Preference

Read Preference 决定 MongoDB 客户端从哪个节点上读取数据。

下图为不同的 Read Preference Driver 端所读取的节点。



默认情况下，应用程序将其读取操作指向副本集中的 Primary 节点。

指定 Read Preference 选项时要注意，因为使用异步复制，复制延迟会导致 Secondary 上的数据可能不是最新的。

默认情况下，复制集的所有读请求都发到 Primary，Driver 可通过设置 Read Preference 来将读请求路由到其他的节点上。

- Primary：默认规则，所有读请求发到 Primary。
- PrimaryPreferred：Primary 优先，如果 Primary 不可达，则请求 Secondary。
- Secondary：所有的读请求都发到 Secondary。
- SecondaryPreferred：Secondary 优先，当所有 Secondary 不可达时，请求 Primary。
- nearest：读请求发送到最近的可达节点上（通过 ping 探测得出最近的节点）。

maxStalenessSeconds

从 3.4 版本起，新增该参数。由于网络拥塞、磁盘吞吐量低、运行时间长等原因，副本集成员可能会落后于主节点。maxStalenessSeconds 选项允许你配置从最大延迟多少的 Secondary 节点上读取数据，当 Secondary 服务器的复制延迟超过 maxStalenessSeconds 时，客户端将停止将其用于读取操作。

6.13 副本集环境搭建

环境介绍：

```
HE1@192.168.1.248 主 port=27017
HE2@192.168.1.249 从 port=27017
HE4@192.168.1.251 从 port=27017
```

副本集最多有 12 个成员（从 3.0 版本后最多支持 50 个成员），但最多 7 个成员有投票权。

在待部署机器 HE1、HE2、HE4 上执行如下命令：

```
[root@HE1 ~]# echo 'PATH=$PATH:/home/work/mongodb/bin' >> /etc/profile
[root@HE1 ~]#source /etc/profile
[root@HE1 ~]#tar xvf percona-server-mongodb-3.2.12-3.2-centos6-x86_64.tar.gz -C /home/work/
[root@HE1 ~]#mkdir -p /home/work/mongodb/data/db_27017
[root@HE1 ~]#mkdir -p /home/work/mongodb/etc
[root@HE1 ~]#mkdir -p /home/work/mongodb/log
[root@HE1 ~]#mkdir -p /home/work/mongodb/tmp
```

```
    [root@HE1      ~]#mv      /home/work/percona-server-mongodb-3.2.12-3.2/bin
/home/work/mongodb/
    [root@HE1 ~]# chown -R work. /home/work/mongodb
    生成 keyfile
    [root@HE1       ~]#       openssl       rand       -base64       756       >
/home/work/mongodb/etc/mongodb-keyfile-benchmark
    [root@HE1 ~]# chmod 400 /home/work/mongodb/etc/mongodb-keyfile-benchmark
```

scp keyfile 文件到每个待部署机器对应目录上：

```
    [root@HE1 ~]# scp -rp /home/work/mongodb/etc/mongodb-keyfile-benchmark
work@HE2:/home/work/mongodb/etc/
```

编写配置文件：

配置文件的空格和大小写均会导致起库失败，这一点需要注意。

```
    [root@HE1 ~]# vi /home/work/mongodb/etc/mongodb_27017.conf
    systemLog:
      destination: file
      path: "/home/work/mongodb/log/mongodb_27017.log"
      logAppend: true

    #net Options
    net:
      port: 27017
      maxIncomingConnections: 10240
      http:
          enabled: false
          JSONPEnabled: false
          RESTInterfaceEnabled: false

    #security Options
    security:
      authorization: 'enabled'
      keyFile: /home/work/mongodb/etc/mongodb-keyfile-benchmark
      clusterAuthMode: "keyFile"

    #storage Options
```

```
storage:
  engine: "wiredTiger"
  dbPath: /home/work/mongodb/data/db_27017
  indexBuildRetry: true
  journal:
    enabled: true
    commitIntervalMs: 100
  wiredTiger:
    engineConfig:
      cacheSizeGB: 1 #根据内存大小给一般给 50%~70%内存
      journalCompressor: "snappy"
      directoryForIndexes: true
    collectionConfig:
      blockCompressor: "snappy"
    indexConfig:
      prefixCompression: true

#replication Options
replication:
   oplogSizeMB: 40960 #40GB
   replSetName: heleitest #副本集名称

#operationProfiling Options
operationProfiling:
  slowOpThresholdMs: 500
  mode: "slowOp"

processManagement:
  fork: true
```

切换 mongo 用户，起库：

```
 [root@HE1 ~]# su - work
[work@HE1 ~]$ numactl --interleave=all /home/work/mongodb/bin/mongod -f
/home/work/mongodb/etc/mongodb_27017.conf
[work@HE1 ~]$ mongo --host 127.0.0.1 --port 27017
```

```
[root@HE1 ~]# su - work
[work@HE1 ~]$ numactl --interleave=all /home/work/mongodb/bin/mongod -f /home/work/mongodb/etc/mongodb_27017.conf
about to fork child process, waiting until server is ready for connections.
forked process: 2825
child process started successfully, parent exiting
[work@HE1 ~]$ mongo --host 127.0.0.1 --port 27017
Percona Server for MongoDB shell version: 3.2.12-3.2
connecting to: 127.0.0.1:27017/test
Welcome to the Percona Server for MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        https://www.percona.com/doc/percona-server-for-mongodb
Questions? Try the support group
        https://www.percona.com/forums/questions-discussions/percona-server-for-mongodb
> db.version()
3.2.12-3.2
```

转为复制集架构：

```
    >config={_id:'heleitest',members:[{_id:0,host:'192.168.1.248:27017'},{_i
d:1,host:'192.168.1.249:27017'},{_id:2,host:'192.168.1.251:27017'}]}
    >rs.initiate(config)
```

```
> config={_id:'heleitest',members:[{_id:0,host:'192.168.1.248:27017'},{_id:1,host:'192.168.1.249:27017'},{_id:2,host:'192.168.1.251:27017'}]}
{
        "_id" : "heleitest",
        "members" : [
                {
                        "_id" : 0,
                        "host" : "192.168.1.248:27017"
                },
                {
                        "_id" : 1,
                        "host" : "192.168.1.249:27017"
                },
                {
                        "_id" : 2,
                        "host" : "192.168.1.251:27017"
                }
        ]
}
> rs.initiate(config)
{ "ok" : 1 }
heleitest:OTHER>
heleitest:SECONDARY>
heleitest:PRIMARY>
```

执行完上述命令可以看到，几秒后该节点从 OTHER 状态转变为 SECONDARY 状态，即转变为 PRIMARY 主库。

日志中可以看到创建 oplog 和该节点成员从 STARTUP2 状态转变为 SECONDARY 的过程：

```
    2018-01-07T20:25:12.455-0800 I REPL     [conn3] ******
    2018-01-07T20:25:12.455-0800 I REPL     [conn3] creating replication oplog
of size: 40960MB...
    2018-01-07T20:25:12.456-0800  I  NETWORK     [initandlisten]  connection
accepted from 192.168.1.249:61142 #4 (2 connections now open)
    2018-01-07T20:25:12.469-0800  I  NETWORK     [initandlisten]  connection
accepted from 192.168.1.251:39875 #5 (3 connections now open)
    2018-01-07T20:25:12.470-0800 I ACCESS   [conn4] Successfully authenticated
as principal __system on local
```

```
    2018-01-07T20:25:12.481-0800    I    STORAGE        [conn3]    Starting
WiredTigerRecordStoreThread local.oplog.rs
    2018-01-07T20:25:12.481-0800 I STORAGE  [conn3] The size storer reports that
the oplog contains 0 records totaling to 0 bytes
    2018-01-07T20:25:12.481-0800  I  STORAGE  [conn3]  Scanning  the  oplog  to
determine where to place markers for truncation
    2018-01-07T20:25:12.483-0800 I ACCESS   [conn5] Successfully authenticated
as principal __system on local
    2018-01-07T20:25:12.488-0800 I REPL     [conn3] ******
    2018-01-07T20:25:12.499-0800 I REPL     [ReplicationExecutor] New replica
set config in use: { _id: "heleitest", version: 1, protocolVersion: 1, members:
[ { _id: 0, host: "192.168.1.248:27017", arbiterOnly: false, buildIndexes: true,
hidden: false, priority: 1.0, tags: {}, slaveDelay: 0, votes: 1 }, { _id: 1, host:
"192.168.1.249:27017", arbiterOnly: false, buildIndexes: true, hidden: false,
priority:  1.0,  tags:  {},  slaveDelay:  0,  votes:  1  },  {  _id:  2,  host:
"192.168.1.251:27017", arbiterOnly: false, buildIndexes: true, hidden: false,
priority: 1.0, tags: {}, slaveDelay: 0, votes: 1 } ], settings: { chainingAllowed:
true,     heartbeatIntervalMillis:     2000,     heartbeatTimeoutSecs:     10,
electionTimeoutMillis: 10000, getLastErrorModes: {}, getLastErrorDefaults: { w:
1, wtimeout: 0 }, replicaSetId: ObjectId('5a52f2a86ed6ebb3483e6a51') } }
    2018-01-07T20:25:12.500-0800 I REPL     [ReplicationExecutor] This node is
192.168.1.248:27017 in the config
    2018-01-07T20:25:12.500-0800 I REPL     [ReplicationExecutor] transition to
STARTUP2
    2018-01-07T20:25:12.500-0800  I  REPL       [conn3]  Starting  replication
applier threads
    2018-01-07T20:25:12.500-0800  I  REPL       [ReplicationExecutor]  Member
192.168.1.251:27017 is now in state STARTUP
    2018-01-07T20:25:12.500-0800  I  REPL       [ReplicationExecutor]  Member
192.168.1.249:27017 is now in state STARTUP
    2018-01-07T20:25:12.502-0800 I REPL     [ReplicationExecutor] transition to
RECOVERING
    2018-01-07T20:25:12.503-0800 I REPL     [ReplicationExecutor] transition to
SECONDARY
```

创建一个超管用户：

```
    >use admin
```

```
>db.createUser(
    {
        user: "sys_admin",
        pwd: "MANAGER",
        roles: [ { role: "root", db: "admin" } ]
      }
  )
```

可以使用 db.auth 来进行认证：

```
>db.auth('sys_admin','MANAGER')
```

```
heleitest:PRIMARY> use admin
switched to db admin
heleitest:PRIMARY> db.createUser(
...     {
...         user: "sys_admin",
...         pwd: "MANAGER",
...         roles: [ { role: "root", db: "admin" } ]
...       }
...   )
Successfully added user: {
        "user" : "sys_admin",
        "roles" : [
                {
                        "role" : "root",
                        "db" : "admin"
                }
        ]
}
heleitest:PRIMARY> db.auth('sys_admin','MANAGER')
1
```

也可以在登录数据库时就进行认证：

```
[mongo@HE1 ~]$ mongo 127.0.0.1:27017/admin -u sys_admin -p MANAGER
```

使用副本集方式登录：

```
[mongo@HE1 ~]$/home/work/mongodb/bin/mongo --host
heleitest/192.168.1.248:27017,192.168.1.249:27017,192.168.1.251:27017 -u
sys_admin -p MANAGER --authenticationDatabase admin
```

能够看到副本集状态已经完成，其中 192.168.1.248:27017 为 PRIMARY，192.168.1.249:27017 和 192.168.1.251:27017 为 SECONDARY：

```
heleitest:PRIMARY> rs.status()
{
        "set" : "heleitest",
        "date" : ISODate("2018-01-08T04:32:21.841Z"),
        "myState" : 1,
        "term" : NumberLong(1),
        "heartbeatIntervalMillis" : NumberLong(2000),
        "members" : [
            {
                  "_id" : 0,
                  "name" : "192.168.1.248:27017",
                  "health" : 1,
                  "state" : 1,
                  "stateStr" : "PRIMARY",
                  "uptime" : 1924,
                  "optime" : {
                  "ts" : Timestamp(1515385876, 4),
                  "t" : NumberLong(1)
            },
            "optimeDate" : ISODate("2018-01-08T04:31:16Z"),
            "electionTime" : Timestamp(1515385522, 1),
            "electionDate" : ISODate("2018-01-08T04:25:22Z"),
            "configVersion" : 1,
            "self" : true
        },
        {
            "_id" : 1,
            "name" : "192.168.1.249:27017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 429,
            "optime" : {
                "ts" : Timestamp(1515385876, 4),
                "t" : NumberLong(1)
      },
       "optimeDate" : ISODate("2018-01-08T04:31:16Z"),
```

```
        "lastHeartbeat" : ISODate("2018-01-08T04:32:21.492Z"),
        "lastHeartbeatRecv" : ISODate("2018-01-08T04:32:21.149Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "192.168.1.248:27017",
        "configVersion" : 1
},
{
        "_id" : 2,
        "name" : "192.168.1.251:27017",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 429,
        "optime" : {
        "ts" : Timestamp(1515385876, 4),
        "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2018-01-08T04:31:16Z"),
        "lastHeartbeat" : ISODate("2018-01-08T04:32:21.492Z"),
        "lastHeartbeatRecv" : ISODate("2018-01-08T04:32:21.149Z"),
        "pingMs" : NumberLong(0),
        "syncingTo" : "192.168.1.248:27017",
        "configVersion" : 1
    }
  ],
  "ok" : 1
}
```

查看复制集的从库状态信息（延迟、成员）：

```
heleitest:PRIMARY> rs.printSlaveReplicationInfo()
source: 192.168.1.249:27017
      syncedTo: Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
      0 secs (0 hrs) behind the primary
source: 192.168.1.251:27017
      syncedTo: Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
      0 secs (0 hrs) behind the primary
```

```
heleitest:PRIMARY> rs.printSlaveReplicationInfo()
source: 192.168.1.249:27017
        syncedTo: Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
        0 secs (0 hrs) behind the primary
source: 192.168.1.251:27017
        syncedTo: Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
        0 secs (0 hrs) behind the primary
```

查看复制集的 oplog 信息：

```
heleitest:PRIMARY> rs.printReplicationInfo()
configured oplog size:   40960MB
log length start to end: 364secs (0.1hrs)
oplog first event time:  Sun Jan 07 2018 20:25:12 GMT-0800 (PST)
oplog last event time:   Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
now:                     Sun Jan 07 2018 20:35:06 GMT-0800 (PST)
```

```
heleitest:PRIMARY> rs.printReplicationInfo()
configured oplog size:   40960MB
log length start to end: 364secs (0.1hrs)
oplog first event time:  Sun Jan 07 2018 20:25:12 GMT-0800 (PST)
oplog last event time:   Sun Jan 07 2018 20:31:16 GMT-0800 (PST)
now:                     Sun Jan 07 2018 20:35:06 GMT-0800 (PST)
heleitest:PRIMARY>
```

6.14 配置延迟

从主库执行：

```
cfg = rs.conf()
cfg.members[2].priority = 0
cfg.members[2].hidden = true
cfg.members[2].slaveDelay = 21600
rs.reconfig(cfg)
```

注意复制集版本要一致，而且 rs.reconfig 某些情况下可能会强制主库降级，降级期间会花费 10～20 秒。在主库执行的时候确认 members[n]，n 为不要指向主库。延时必须设置：

```
priority=0
hidden=true
```

延时时间低于 oplog window。

members[n].votes=1：默认具备投票。

6.15　从 2.6 版本升级至 3.0 版本

限制

要升级到 3.0 版本，首先现有集群必须是 2.6 版本。如果是早期的版本，则必须先升级到 2.6 版本才可以升级到 3.0 版本。如果 2.6 版本的集群已经开启了认证模式，升级前需进行 authSchema 版本升级，详见认证相关章节。

准备

在开始升级之前，请参阅 MongoDB 3.0 版本文档中的兼容性更改，以确保应用程序和部署与 MongoDB 3.0 版本兼容，在开始升级之前解决部署中的不兼容问题。

升级 MongoDB 之前，请先在临时环境中测试应用程序，以确保升级顺利进行。

降级版本的限制

升级到 3.0 版本后，只能降级到 2.6.8 版本或更高版本。

> **注意：**避免重新配置包含不同 MongoDB 版本成员的副本集，因为 MongoDB 版本中的权限验证规则可能会有所不同。。

低版本升级先决条件

要将副本集升级到 3.0 版本，所有副本集成员必须运行 2.6 版本。要从早期的 MongoDB 版本升级副本集，请先将副本集的所有成员升级到最新的 2.6 系列版本，然后按照以下步骤从 MongoDB 2.6 升级到 3.0 版本。

6.15.1　升级过程

先升级副本集中的一个 Secondary

关闭 mongod 实例，并用 3.0 二进制文件的 bin 目录代替 2.6 版本二进制文件的 bin 目录。

重新启动成员并等待成员恢复到 Secondary 状态，然后升级下一个 Secondary 成员。要检查成员的状态，可在 mongo Shell 中发出 rs.status()。

stepdown 副本集中的 Primary

将 mongo Shell 连接到 Primary 服务器并使用 rs.stepDown()来降级主服务器并强制其他节点选举新的 Primary 服务器。

升级 Primary 服务器

当 rs.status()显示原 Primary 节点已经变为 Secondary，新的 Primary 已经被选举出来的时候，开始升级原 Primary 节点：

关闭数据库，并用 3.0 二进制文件的 bin 目录代替 2.6 二进制文件的 bin 目录。

启动原 Primary 节点。

6.15.2 关于认证

2.6 版本开启认证模式要升级到 3.0 版本，必须先升级 authSchema。

运行 authSchemaUpgrade 并升级到 3.0 版本后，无法降级到 2.6 版本并开启认证模式。

查看 authSchema 版本是不是 3.0（at least）：

```
use admin
db.system.version.find( { _id: "authSchema" })
```

升级 SchemaUpgrade，如果是复制集，则在主库执行该命令：

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1 });
```

6.15.3 变更存储引擎

必须使用 MongoDB 3.0 或更高版本才能使用 WiredTiger 存储引擎。如果从早期版本的 MongoDB 升级，可参阅升级到 MongoDB 3.0 或 MongoDB 3.2 的指导，然后继续更改存储引擎。

在启用新的 WiredTiger 存储引擎之前，请确保所有副本集/分片集群成员至少运行 MongoDB 2.6.8 版本，最好是 3.0.0 版本或更高版本。副本集可以具有不同存储引擎的成员，因此，可以更新成员并以滚动的方式使用 WiredTiger 存储引擎。在更改所有成员使用 WiredTiger 之前，可能希望在一段时间内运行混合存储引擎，但是性能会因工作量而异。

在 configurefile 中添加 engine: "WiredTiger"，并在更换 dbpath 后重启实例。

由于 dbpath 是新目录，副本集会执行 initial sync 进行重新同步，重新同步的时间取决于数据量的大小和网络情况。

为使用 WiredTiger 存储引擎运行的新 mongod 实例准备数据目录，mongod 必须具有此目录的读写权限。可以删除已停止辅助成员的当前数据目录的内容，也可以完全创建新的目录。

WiredTiger 的 mongod 不会从使用不同的存储引擎创建的数据文件开始。

6.15.4　Driver 兼容性

绝大部分 dirver 版本只要支持 3.0 版本也会支持 3.2 版本。

更多 driver 兼容性参见：

https://docs.mongodb.com/ecosystem/drivers/driver-compatibility-reference/。

6.16　从 3.2 版本升级至 3.4 版本

限制

要升级到 3.4 版本，首先现有集群必须是 3.2 版本。如果是早期的版本，则必须先升级到 3.2 版本才可以升级到 3.4 版本。

准备

在开始升级之前，请参阅 MongoDB 3.4 文档中的兼容性更改，以确保应用程序和部署与 MongoDB 3.4 版本兼容。在开始升级之前解决部署中的不兼容问题。

在升级 MongoDB 之前，请先在临时环境中测试应用程序，以确保升级顺利进行

降级版本的限制

升级到 3.4 版本后，不能降级到 3.2.7 版本或更早版本，只能降级到 3.2.8 版本或更高版本。

> **注意：**避免重新配置包含不同 MongoDB 版本成员的副本集，因为 MongoDB 版本中的权限验证规则可能会有所不同。

低版本升级先决条件

要将副本集升级到 3.4 版本，所有副本集成员必须运行 3.2 版本。要从早期的 MongoDB 版本升级副本集，可先将副本集的所有成员升级到最新的 3.2 系列版本，然后按照下节的步骤从 MongoDB 3.2 版本升级到 3.4 版本。

6.16.1　升级过程

先升级副本集中的一个 Secondary

关闭 mongod 实例，并用 3.4 版本二进制文件的 bin 目录代替 3.2 版本二进制文件的 bin 目录。

重新启动成员并等待成员恢复到 Secondary 状态，然后升级下一个 Secondary 成员。要检查成员的状态，请在 mongo Shell 中发出 rs.status()。

stepdown 副本集中的 Primary

将 mongo Shell 连接到 Primary 服务器并使用 rs.stepDown()来降级主服务器并强制其他节点选举新的 Primary 服务器。

升级 Primary 服务器

当 rs.status()显示原 Primary 节点已经变为 Secondary 时，新的 Primary 已经被选举出来的时候，开始升级原 Primary 节点：

关闭数据库，并用 3.4 版本二进制文件的 bin 目录代替 3.2 版本二进制文件的 bin 目录。

启动原 Primary 节点。

6.16.2 启用不向下兼容的 3.4 版本功能

此时，可以运行 3.4 版本，而不使用与 3.2 版本不兼容的 3.4 版本的功能。

要启用这些 3.4 版本的功能，可将功能兼容版本设置为 3.4。

启用这些不向下兼容的功能会使降级过程复杂化。

建议在升级后，3.4 版本的这些功能在一段时间内先不使用，以确保稳定性。当确认应用稳定运行，且需要 3.4 版本这些不向下兼容的功能时，利用如下命令启用这些功能。主节点运行：

```
db.adminCommand( { setFeatureCompatibilityVersion:"3.4" } )
```

6.16.3 升级发现 infoMessage 异常

```
"members" : [
        {
                "_id" : 0,
                "name" : "c3-xxx-mgdb01.bj:27020",
                "health" : 1,
                "state" : 2,
                "stateStr" : "SECONDARY",
                "uptime" : 121,
                "optime" : {
                        "ts" : Timestamp(1501747251, 50),
```

```
            "t" : NumberLong(5)
        },
        "optimeDate" : ISODate("2017-08-03T08:00:51Z"),
        "infoMessage" : "could not find member to sync from",
        "configVersion" : 5,
        "self" : true
}
```

这时升级成功了，stateStr 状态也是 Secondary，但 infoMessage 出现异常，这个异常有误导是可以忽略的。出现这个错误的原因是主库没有写入，主从状态是一致的，oplog 不是最新的，因此会有这个提示。一旦主库有写入这个信息就会消失，预计在 MongoDB 3.5 中将得到修复。

6.17　分片

Sharding

分片是一种在多台机器上分配数据的方法。MongoDB 使用分片架构有助于管理数量非常大的数据集和高吞吐量操作的集群。

大数据量和高吞吐量的业务情况对单台服务器来讲具备很大的挑战性。例如，高查询率可能耗尽服务器的 CPU 容量。工作集大小如果超过了系统内存，那么压力就会给到磁盘上，这对 I/O 来讲不是我们所希望的。

解决系统业务量增长有两种方法：垂直缩放和水平缩放。

垂直缩放采用增加单个服务器容量的方法，例如，使用更强大的 CPU，增加更多 RAM 或增加存储空间量。但无论从技术上还是经济上考虑，都会限制单台机器的性能。此外，性能不是无限扩张的，会有一个瓶颈值，因此垂直缩放是不经济而且有上限的。

水平缩放涉及将系统数据集和负载分配到多个服务器上，添加额外的服务器以根据需要增加容量。虽然单台机器的整体速度或容量可能并不高，但每台机器都会处理整个工作负载的一个子集，可能比单个高速大容量服务器提供更高的效率。扩展部署的容量只需要按需添加额外的服务器，与单个机器的高端硬件相比，总体成本可能更低，这种办法的确是增加了部署和维护的复杂性。

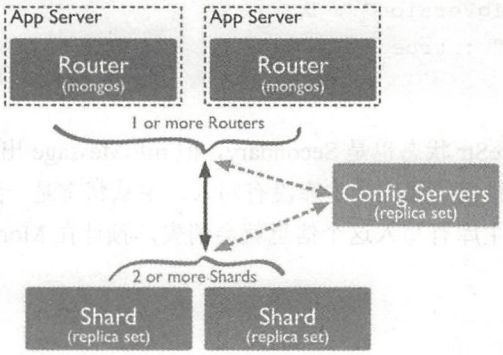
MongoDB 支持通过分片进行水平缩放。

MongoDB 分片集群由以下组件组成。

- Shard：每个 Shard 包含分片数据的一个子集，每个分片都可以作为副本集进行部署。
- mongos：mongos 作为查询路由器，提供客户端应用程序和分片集群之间的接口。

- configserver：配置服务器存储集群的元数据和配置设置。从 MongoDB 3.4 开始，必须将配置服务器部署为副本集（CSRS）。

下图描述了分片集群内组件的交互。



MongoDB 在集合级别分割数据，将集合数据分布到集群中的分片上。

Shard Keys

为了对一个集合中的文档进行分割，MongoDB 使用 shard key 对集合进行分割，shard key 由目标集合中每个文档的不可变字段组成。

分片集合时选择的 shard key 在分片后不能改变，且分片集合只能有一个 shard key。

如果要分割非空集合，则集合必须具有以分片键开头的索引。对于空集合，如果集合尚不具有指定分片键的适当索引，则 MongoDB 会自动为其创建索引。

分片密钥的选择会影响分片群集的性能、效率和可伸缩性。再好的机器硬件，如果 shard key 的选择不合理，也一样会遇到瓶颈。分片键及其支持索引的选择也会影响集群可以使用的分片策略。

Chunks

MongoDB 将分片数据分成块，每个块具有基于分片键左闭右开的范围值。

MongoDB 使用分片的 balancer 将分块移入分片集群中的分片，balancer 会试图在集群中的所有 Shard 上实现 Chunk 的均衡。

分片的优点

reads/writes

MongoDB 将读写负载分布在分片集群中的分片上，允许每个分片处理集群操作的一个子集。读取和写入工作负载可以通过添加更多的 Shard 在集群中水平扩展。

对于包含分片键或复合分片键的前缀的查询，mongos 可以将查询定位到特定的分片或分片集中，这些有针对性的操作通常比向集群中的每个分片查询更高效。

存储容量

分片将数据分布到集群中的分片上，允许每个分片包含总集群数据的一个子集。随着数据集的增长，新加入的 Shard 会分担数据的存储量。

高可用性

即使一个或多个分片不可用，分片集群也可以继续执行部分读取/写入操作。虽然在停机期间无法访问不可用分片上的数据子集，但是直接访问可用分片的读取或写入操作仍然可以成功。

从 MongoDB 3.2 开始，你可以将 config server 部署为副本集。只要大多数副本集可用，使用 config server 副本集（CSRS）的分片集群就可以继续处理读取和写入操作。在 3.4 版本中，MongoDB 取消了对 SCCC 配置服务器的支持。

在生产环境中，应将 Shard 部署为副本集，从而提供更高的冗余性和可用性。

分片之前的注意事项

分片的集群配置之初，就需要仔细的规划、执行和维护。

仔细考虑选择 shard key 对于确保集群性能和效率是必要的。在分片之后你不能改变 shard key，也不能将 Shard 集合进行合并。

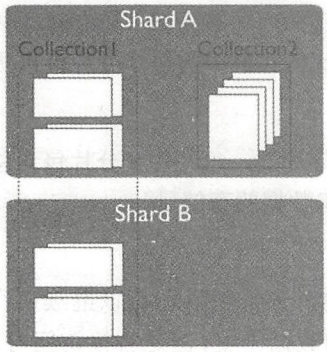
Sharding 后有一定的操作要求和限制。

如果查询不包括 shard key 或使用符合 shard key 开头的索引，则 mongos 执行整个分片扫描操作，查询分片架构中的所有分片，这种操作耗时非常长。

6.17.1　分片和非分片集合

数据库可以有分片和非分片集合的混合，分片集合被分区并分布在集群中的分片上。Unsharded 集合存储在主分片上，每个数据库都有自己的主分片。

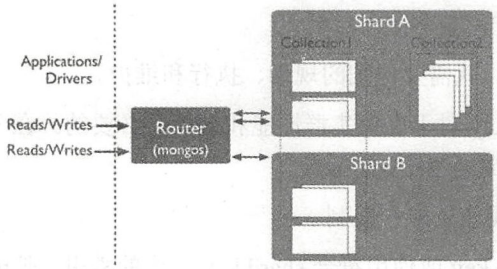
下图是一个最简单的两个分片架构，可以看出所有未开启分片的集合都会存储在主分片上。



连接到分片群集

必须连接到 mongos 才能与分片群集中的集合进行交互，包括分片和不分片的集合。客户端不应连接到单个分片机器上去执行读取或写入操作。

下图是一个简单的业务，通过 Dirvers 访问 mongos 来对 sharding 数据进行读写。



分片策略

MongoDB 支持两种分片策略，用于在分片集群之间分发数据。

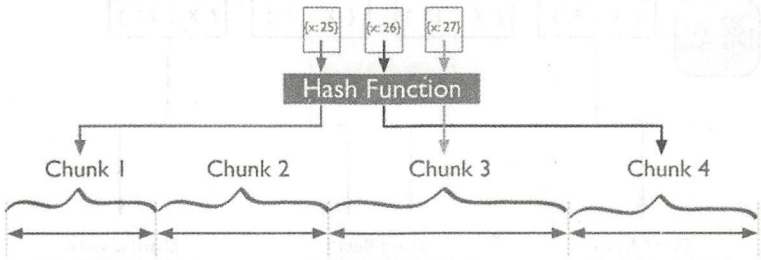
（1）Hash 分片

Hash 分片涉及计算分片键字段值的散列，然后根据散列的分片键值为每个块分配一个范围。

MongoDB 在使用 Hash 索引解析查询时自动计算散列值，而应用程序不需要计算哈希值。

从下图可以看出，Hash 分片使用 Hash 索引在 sharding 中分区数据。Hash 索引计算单个字段的哈希值作为索引值，此值用作分片键。

虽然 Hash 分片对范围查询不够友好，但它们的散列值不可能在同一 Chunk 上。因此基于 Hash 值的数据分布有助于更均匀地分布数据，特别是在 shard key 单调变化的数据集中。

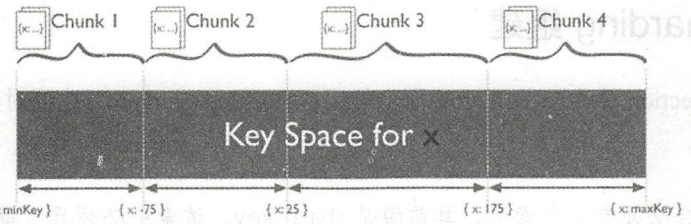


然而，Hash 分布意味着查询范围大，需要扫描所有 Shard，这个效率是比较低的。

（2）range 分片

range 分片根据范围将值放到指定的 Shard 中，每个 Chunk 根据自己的范围存放到对应的 Shard 中。

从下图能够看出，值相近的会放入一个 Chunk 中，如果查询的文档涉及的 Chunk 是连续的，则对于范围查询来说会有较好的效率。



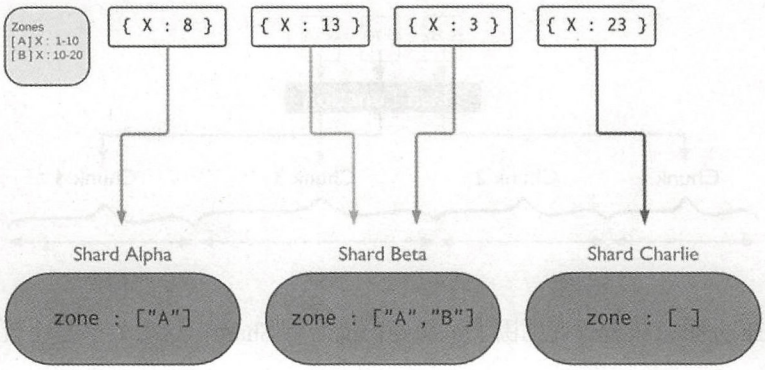
range 型分片使得数据可能驻留在相同的 Chunk 上，这允许有针对性的操作，因为 mongos 可以将对应的操作仅路由到包含该数据的 Shard 上。

范围分片的效率取决于选择的分片键，shard key 选择得不好会导致数据分布不均匀，这可能会使得整个分片集群遇到瓶颈。

Shard 集群中的 zone

在分片群集中，可以基于 shard keys 创建 zone。可以将每个 zone 与群集中的一个或多个 Shard 相关联，一个 Shard 可以与多个非冲突 zone 相关联。在一个 balance 的集群中，MongoDB 只将 Chunk 移动到对应的 zone 里。

如下图所示，一个 zone 可以包含多个范围，一个 zone 可以划分到多个 Shard 中，zone 覆盖的范围始终是左闭右开的。



定义 zone 时，必须包含 shard key。如果使用复合 shard key，shard key 列必须作为复合索引的前缀（类似 MySQL 的最左前缀原则）。

选择 shard key 时，要仔细考虑以后如何使用 shard key，因为在分片之后无法更改 shard key。

通常，zone 用于改善跨越多个数据中心分片集群数据的位置。

6.17.2 Sharding 组建

将 shardCollection 命令与 collation 配合使用：{locale："simple"}选项对具有默认排序规则的集合进行分片。

> **要求：**集合必须有一个索引，其前缀是 shard key。该索引必须具有排序规则{locale："simple"}。使用排序规则创建新的集合时，请确保在分割集合之前满足这些条件。

MongoDB 分片集群由以下组件组成。

- Shard：每个 Shard 包含分片数据的一个子集，每个分片都可以作为副本集进行部署。
- mongos：mongos 作为查询路由器，提供客户端应用程序和分片集群之间的接口。
- config server：配置服务器存储集群的元数据和配置设置。从 MongoDB 3.4 开始，必须将配置服务器部署为副本集（CSRS）。

下面介绍生产环境配置过程。

在生产集群中，确保数据是冗余的，并确保系统具有高可用性。生产分片集群部署一般为：

- 将 config server 部署为 3 个成员副本集；
- 将每个 Shard 部署为 3 个成员副本集；
- 部署一个或多个 mongos。

在可能的情况下，可以考虑将副本集中的一个成员放到异地，以应对灾难恢复。

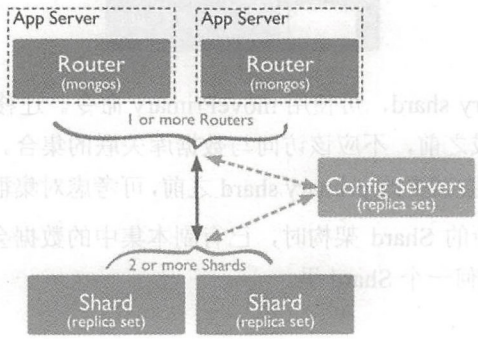
分片架构需要至少两个 Shard。

部署多个 mongo 以提高可用性和可伸缩性，一个常见的方法是在每个应用程序服务器上放置一个 mongos。在每个应用程序服务器上部署一个 mongos 路由器可减少应用程序和路由器之间的网络延迟。

或者可以在每个分片主节点上放置一个 mongos 路由器，应用程序使用连接字符串列出每个分片主节点的所有主机名。然后，MongoDB 驱动程序确定每个 mongos 的网络延迟，并在所设置的延迟窗口内的路由器上随机进行负载平衡。这可以确保主 Shard 和 mongos 的服务器有足够的 CPU 和内存来提供服务。

可以随意部署 mongos 的数量，但是由于 mongos 经常与 config server 进行通信，因此在增加 mongos 数量时要密切监视 config server 的性能。如果你看到性能下降，那么在部署中限制 mongos 的数量可能会有所帮助。

如下图所示，mongos 可以有多个，而 config server 是一个副本集架构，后端的 Shard 分片至少有两个，这是最简单的 sharding 架构。



6.17.3　Shard

Shard 包含分片集群的分片数据的子集，集群的 Shard 一起保存整个集群的数据集。

Shard 应作为副本集来提供冗余和高可用性。

用户、客户端或应用程序只有维护 Shard 集群时，才需要直连 Shard 里的 mongod。

在单个 Shard 上执行查询只返回数据的一个子集，用户应该连接到 mongos 执行集群级操作，包括读取或写入操作。

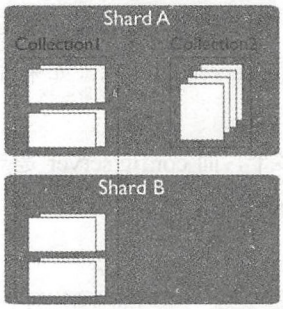
MongoDB 不保证任何两个连续的 Chunk 都驻留在一个 Shard 上。

primary shard

分片群集中的每个数据库都有一个主分片，用于存放该数据库的所有未分片集合。每个数据库都有自己的主分片，primary shard 和副本集中的 Primary 成员不是一回事。

mongos 在执行 create databases 命令时，会选择集群中具有最少数据量的 Shard 来作为 primary shard。mongos 使用 listDatabase 命令返回的 totalSize 字段作为选择条件的一部分。

如下图所示，分片群集中的每个数据库都有一个主分片。后期也可以通过命令来进行主分片的迁移。



要更改数据库的 primary shard，可使用 movePrimary 命令。迁移 primary shard 的过程可能需要很长时间，并且在完成之前，不应该访问与数据库关联的集合。迁移数据量大时，迁移可能会影响整个群集操作。在尝试更改 primary shard 之前，可考虑对集群操作和网络负载的影响。

使用已有副本集部署新的 Shard 架构时，已有副本集中的数据会被保留，而新创建的数据则会出现在 shard 架构的任何一个 Shard 里。

shard status

使用 mongo Shell 中的 sh.status()方法查看集群的概况，能够看到哪个 Shard 是 primary shard，以及 Chunk 在整个 Shard 中的分布情况。

6.17.4 Config server

从 3.4 版本开始，不再支持使用已弃用的镜像 mongod 实例作为 config server（SCCC）。在将分片集群升级到 3.4 版本之前，必须将配置服务器从 SCCC 转换为 CSRS(config server 的副本集架构)。

config server 存储分片集群的元数据。元数据反映了分片集群内所有数据和组件的状态和组织，它包括每个分片上的 Chunk 列表及定义 Chunk 的范围。

mongos 实例缓存这些数据，并将读写操作路由到正确的分片上。mongos 在集群发生元数据更改（例如，“块拆分”或添加 Shard）时更新缓存，Shard 也从 config server 读取 Chunk 元数据。

config server 还存储身份验证配置信息，如基于角色的访问控制或集群的内部身份验证设置。

MongoDB 也使用 config server 来管理分布式锁。

每个分片集群都必须有自己的 config server，不要为不同的分片集群使用相同的 config server。

在 config server 上执行的管理操作可能会对分片群集的性能和可用性产生重大影响。根据受影响的 config server 的数量，集群可能会在一段时间内为只读或脱机状态。

Replica Set Config Servers

从 MongoDB 3.2 开始，分片群集的 config server 可以部署为一个副本集（CSRS），而不是 3 个镜像 config server（SCCC），使用 config server 的副本集提高了整个 config server 的一致性。此外，使用 config server 的副本集可使分片集群具有 3 个以上 config server，因为副本集最多可包含 50 个成员。要将 config server 部署为副本集，配置服务器必须运行 WiredTiger 存储引擎。

在 3.4 版本中，MongoDB 取消了对 SCCC 配置服务器的支持。

部署 config server 时，以下限制适用于副本集配置：

- 不能有仲裁节点。
- 不能有延迟的成员。
- 必须能构建索引（即没有成员应该将 buildIndexes 设置为 false）。

config server 上的读写操作

admin 库和 config 库在 config server 上。

（1）config server 写入相关

admin 数据库包含与认证和授权相关的集合。

config server 包含分片集群元数据的集合。MongoDB 在元数据改变时会将数据写入 config server，例如，块迁移或块拆分之后。

用户应避免在正常运行或维护过程中直接写入配置数据库。

当写入配置服务器时，MongoDB 使用“majority”级别的 write concern。

（2）从 config server 读取

MongoDB 从 admin 数据库读取认证和授权数据及其他内部用途。

MongoDB 在 mongos 启动时或在元数据发生变化（例如块迁移之后）时从 config server 中读取数据，Shard 也从 config server 读取 Chunk 元数据。

从副本集 config server 读取时，MongoDB 使用“majority”级别的 read concern。

config server 的可用性

如果 config server 副本集丢失其主节点，并且不能选择主节点，则该集群的元数据将变为只读状态。仍然可以读取和写入 Shard 中的数据，但不能够执行 Chunk 迁移和 Chunk 拆分，直到 config server 有 Primary 节点选举出来。

在分片集群中，mongod 和 mongos 实例会监视分片集群中的 mongod 副本集和 config server 副本集。

如果所有的 config server 都变得不可用，则集群可能无法使用。为确保 config server 保持可用且完好无损，config server 的备份至关重要。config server 上的数据及存储与整个 Shard 集群的数据相比较小，并且 config server 的负载相对较低。

Sharded Cluster Metadata

config server 将元数据存储在 config 库中。

在 config server 上进行任何维护之前，应始终备份 config 库。

一般来说，不应该直接修改 config 数据库的内容。config 数据库包含以下集合：

```
changelog
chunks
collections
databases
lockpings
locks
mongos
settings
shards
version
```

6.17.5　mongos

MongoDB mongos 实例将查询和写操作路由到分片集群对应的 Shard 中。mongos 是给应用提供的唯一接口，应用程序永远都不应该去连接 config server 或者 mongod。

mongos 通过从 Config server 中缓存元数据来跟踪数据在哪个 Shard 上，并使用元数据将来自应用程序和客户端的操作路由到 mongod 实例中。mongos 只是缓存数据，并不能持久化，并且消耗的系统资源也比较少，可以部署在稍差的机器上。

最常见的做法是，在应用程序服务器上运行部署 mongos 实例，也可以在 Shard 或其他专用资源上部署 mongos 实例。

mongos 的路由和结果

mongos 实例通过以下方式将查询路由到集群：

- 确定必须接收查询的 Shard 列表。
- 在所有目标 Shard 上建立一个游标。

mongos 合并来自每个目标 Shard 的数据并返回结果。某些特殊操作例如 sort，会先在 primary shard 上进行排序操作，之后再返回给 mongos。

在 3.2 版本的新特性中，对于在多个 Shard 上进行 aggregation 操作，如果不需要在数据库的 primary shard 上运行，则可以将结果路由到任何 Shard 中进行合并，以避免 primary shard 的负载过高。

当 shard key 符合索引“最左前缀原则”时，mongos 会在对应的 Shard 上进行操作，而不必扫描整个 shard 集群。

mongos 对不包含 shard key 的查询，需要扫描整个 Shard 的所有数据后才能返回结果。某些包含 shard key 的查询可能仍会导致扫描整个 Shard 的广播操作出现，具体方式取决于集群中的数据分布和查询的选择性。

mongos 如何处理查询模型

（1）Sorting

如果查询的结果未被排序，则 mongos 实例将打开一个结果游标，其结果是来自 Shard 上所有游标的“round robins”。

如果查询使用 sort()指定的排序结果，则 mongos 实例将$orderby 选项传递给 Shard。数据库的 primary shard 接收并执行所有结果的合并排序，然后通过 mongos 将数据返回给客户端。

（2）Limits

如果查询使用 limit()游标方法限制结果集的大小，则 mongos 实例将该限制传递给 Shard，然后在将结果返回给客户端之前，将限制重新应用于结果。

（3）Skips

如果查询使用 skip()游标方法指定要跳过的记录数，则 mongos 不能将跳转传递给 Shard，而是从分片中检索未读取的结果，并在汇总完整结果时跳过适当数量的文档。

（4）Query Isolation

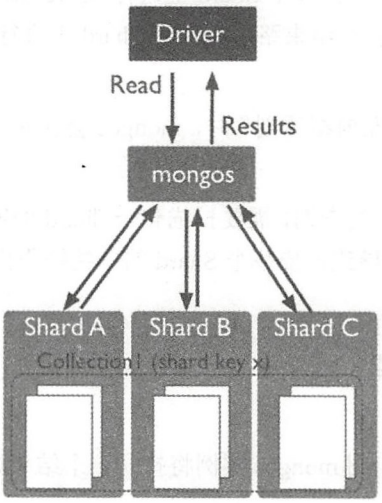
一般来说，分片环境中最快的查询是那些 mongos 路由到单个分片的查询，使用来自 config server 的 shard key 和集群元数据。这些有针对性的操作使用 shard key 来定位满足查询文档的 shard。

对于不包含 shard key 的查询，mongos 必须查询所有分片，等待其响应，然后将结果返回给应用程序。这些“分散/收集”查询耗时会比较长。

广播操作

如果能够通过 shard key 定位到数据在哪个 Shard 上，则能够避免广播操作。

下图显示了 Driver 通过 mongos 访问了每一个 Shard，这样的效率是低下的。

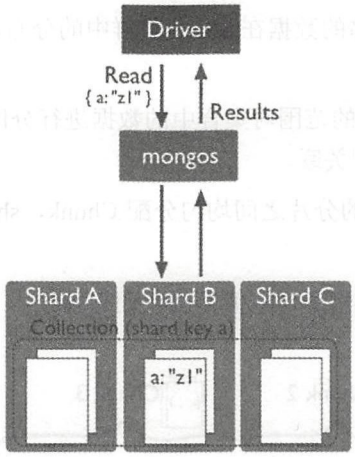


mongos 接收到所有分片的响应后，合并数据并返回结果文档。广播操作的性能取决于集群的总体负载及诸如网络延迟、单个片段负载，以及每个片段返回的文档数量等变量。我们应该尽量避免广播式的扫描，即在集合设计之初的 shard key 上进行选择和规划。

目标操作

mongos 可以将包含分片键或复合分片键前缀的查询路由到特定的分片或一组分片中。mongos 使用分片键值定位其范围，分片键值包含 Chunk，并将查询指向包含该 chunk 的 Shard。

如下图所示，包含分片键的前缀查询会由 mongos 路由至 Shard B，避免查询所有的 Shard，提高效率。



例如，如果分片键是：

```
{a: 1, b: 1, c: 1}
```

则 mongos 程序可以将包含完整 shard key 或以下分片密钥前缀的查询路由到特定分片或一组分片中。

```
{a: 1}
{a: 1, b: 1}
```

所有的 insertOne()操作都以一个分片为目标，insertMany()数组中的每个文档都指向一个分片，但不能保证数组中的所有文档都插入一个分片中。

所有 updateOne()、replaceOne()和 deleteOne()操作都必须在查询文档中包含 shard key 或_id。如果这些方法没有使用 shard key 或_id，MongoDB 会返回一个错误。

根据集群中的数据分布和查询的选择性，mongos 可能仍然执行广播操作来完成这些查询。

索引使用：

如果查询不包括分片键，则 mongos 必须将查询操作发送到所有分片（广播）中，每个分

片依次使用分片索引或另一个更高效的索引来完成查询。

如果查询包含引用分片索引字段和辅助索引字段的多个子表达式，则 mongos 可以将查询路由到特定分片中，以便能够最高效地执行索引。

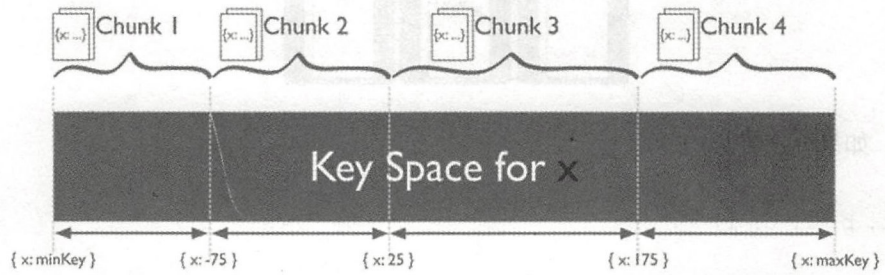
6.17.6 Shard keys

shard key 决定了集合中存储的数据在 Shard 集群中的分布，它是集合中每个文档的索引字段或符合索引的前缀字段。

MongoDB 使用 shard key 值的范围对集合中的数据进行分区，每个范围定义了 shard key 值的非重叠范围，并且与 Chunk 相关联。

MongoDB 会尝试在集群中的分片之间均匀分配 Chunk，shard key 与 Chunk 分布的有效性有直接关系。

下图为一个范围分片的典型例子，取值范围左闭右开。



分割集合后，分片键和分片键值是不可变的，即：

无法为该集合选择不同的 shard key；

无法更新 shard key 字段的值。

指定 shard key

要对集合进行分片，可以使用 sh.shardCollection()：

```
sh.shardCollection(namespace, key)
```

namespace 参数由一个字符串<database>.<collection>组成，指定目标集合的完整名称空间。

key 参数包含字段的文档、该字段的索引和索引方向。

shard key index

所有分片集合都必须有一个支持 shard key 的索引，即索引可以是 shard key 上的索引或 shard key 是索引前缀的复合索引。

如果集合为空，则执行 sh.shardCollection()，将在分片键上创建索引。

如果集合不为空，则必须在使用 sh.shardCollection()之前创建索引。

如果删除分片键的最后一个有效索引，则通过仅在分片键上重新创建索引来进行恢复。

unique index

对于分片集合，只有_id 字段索引、分片键上的索引、或分片键是前缀的复合索引可以是唯一的：

如果其他字段有唯一索引，则不能对该集合进行分片，也无法在分片集合的其他字段上创建唯一索引。

如果集合为空，则 sh.shardCollection()将在分片键上创建唯一索引；如果集合不为空，则必须在使用 sh.shardCollection()之前创建索引。

不能对 Hash 索引指定唯一索引。

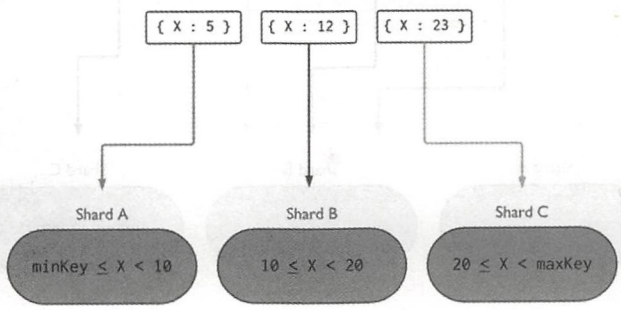
选择一个 shard key

shard key 的选择会影响分片的集群平衡器在可用的分片中创建和分发 chunk 的过程，即影响分片集群内操作的整体效率和性能。

shard key 影响分片集群使用分片策略的性能和效率。

理想的 shard key 允许 MongoDB 在整个集群中平均分配文档。

如下图所示，要考虑 x 在各个范围的分布，如果 X 在 10 到 20 的居多，就会导致分片不均匀，即 Shard B 的数据会远高于 Shard A 和 Shard C，这对性能有很大影响。



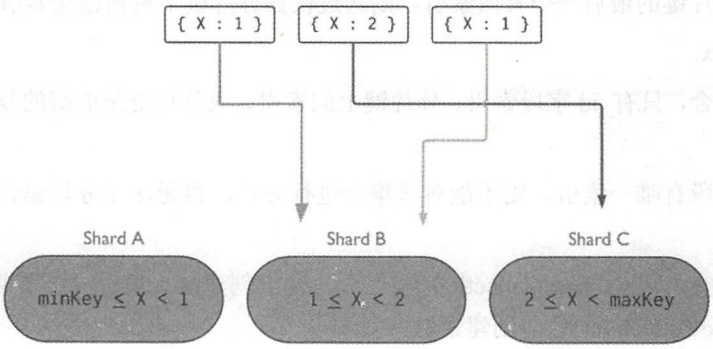
shard key 的基数

分片键的基数决定了平衡器可以创建的最大块数，这可以降低或消除集群中水平缩放的有

效性。

如果每个 Chunk 存的数值是唯一的，shard key 的基数是 4，那么在一个 Shard 中，虽然可以有多个 Chunk，但由于基数是 4，所以这个 Shard 最多只能有 4 个 Chunk。

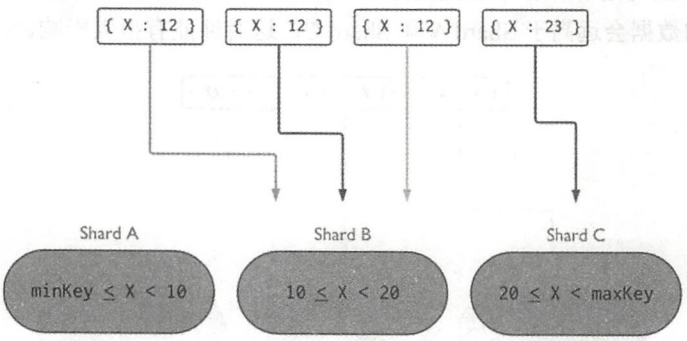
下图显示了使用字段 X 作为分片键的分片集群，如果 X 具有较低的基数，则插入的分布可能看起来类似于以下内容。



shard key 的频率

决定 shard key 时，要考虑到写入的 shard key 频率是否过高，如果大多数文档只包含这些值的一个子集，那么存储这些文档的 Chunk 就成为集群内的一个瓶颈。而且，随着这些大块的成长，它们可能变成不可分割的块，因为它们不能再分裂。

下图显示了使用字段 X 作为分片键的分片集群。如果 X 值的子集高频出现，则插入的分布可能看起来类似于以下内容。



频率低的 shard key 也不能保证在分片集群中数据的均匀分布，shard key 的基数和变化率都和数据分配相关。

如果你的数据模型需要使用具有高频率值的 shard key 进行分片，则可考虑使用唯一或低频率值的复合索引。

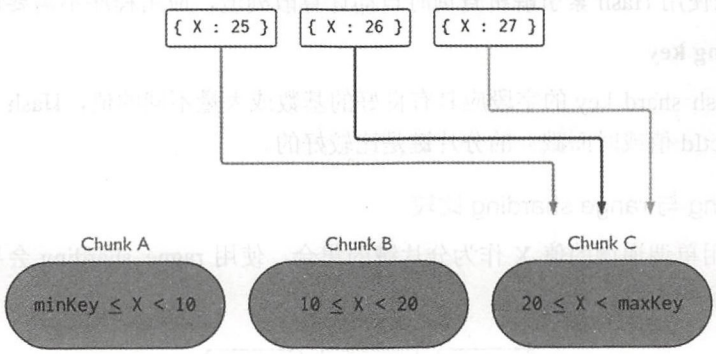
shard key 单调递增或递减

单调增加或减少的 shard key 更有可能导致其中一个 Shard 数据量过大。

发生这种情况是因为每个集群都有一个 Chunk，它捕获一个带有 maxKey 上限的范围，maxKey 总是比所有其他值高。类似地，有一个块捕获一个 minKey 下界的范围，minKey 始终比其他值低。

如果分片键值始终增加，则将所有新插入的键值以 maxKey 作为上限路由到 Chunk 中；如果分片键值始终减小，则将所有新插入的键值以 minKey 作为下限进行路由操作。包含该 Chunk 的碎片将成为写入操作的瓶颈。

下图显示了使用字段 X 作为分片键的分片集群，如果 X 的值单调递增，则插入的分布可能如下所示。



如果分片键值单调减少，则所有插入的键值将转到块 A。

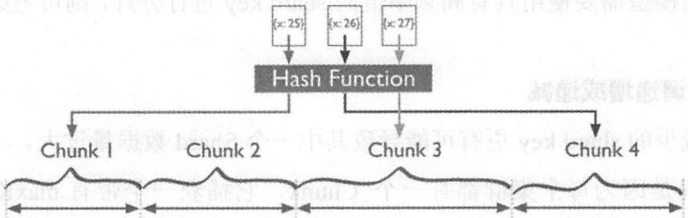
如果数据模型需要在单调变化的密钥上分片，则可考虑使用散列分片。

6.17.7　哈希分片

Hash sharding

Hash 分片使用单个字段的 Hash 索引作为 shard key 来分割分片集群中的数据。

从下图可以看出，Hash 分片使用 Hash 索引在 sharding 中分区数据。Hash 索引计算单个字段的哈希值作为索引值；此值用作分片键。



Hash 分片以范围查询效率变低为代价提供了跨分片分布的更均匀的数据分布。哈希后，具有相近分片键值的文档不太可能位于相同的 Chunk 或 Shard 上，mongos 更可能执行广播操作来完成给定的分段查询。mongos 可以将具有相等匹配的查询定位到单个分片。

如果使用哈希分片键对空集合进行分片，则 MongoDB 会自动为每个分片创建两个空 chunk，以覆盖集群中 Hash shard key 的整个范围。可以控制 MongoDB 使用 numInitialChunks 参数为 shardCollection 创建的 Chunk 数，也可以使用 split 命令手动在空集合上创建 Chunk。

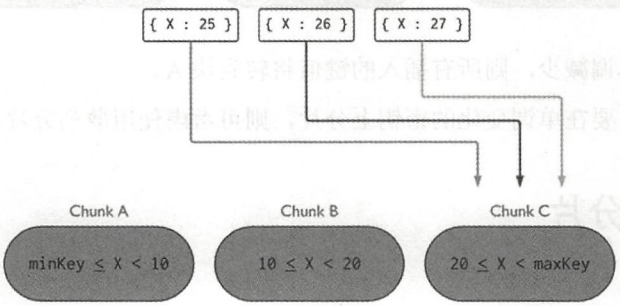
MongoDB 在使用 Hash 索引解析查询时自动计算散列值，应用程序不需要计算哈希值。

Hash sharding key

选择作为 Hash shard key 的字段应具有良好的基数或大量不同的值，Hash key 对于字段单调变化（如 ObjectId 值或时间戳）的分片键是比较好的。

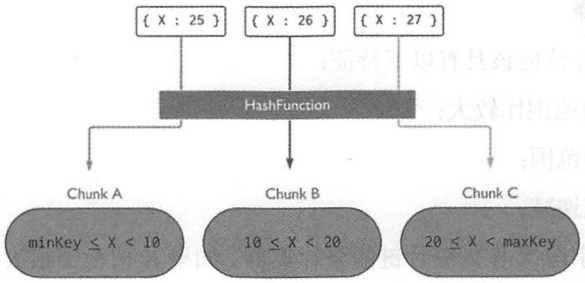
Hash sharding 与 range sharding 比较

给定一个使用单调递增的值 X 作为分片键的集合，使用 ragne sharding 会导致插入的分布类似于如下图所示的内容。



由于 X 的值总是增加，所以具有 maxKey 上限的 Chunk 接收大部分写入，这就导致了数据的分布不均匀，对查询性能造成影响，违背了分片的初衷。

如果使用 Hash sharding，那么会是如下图所示的数据分布。



由于现在数据分布更均匀，对集合的操作会更高效。

将集合进行 Hash 分片

连接到 mongos 上，执行以下命令：

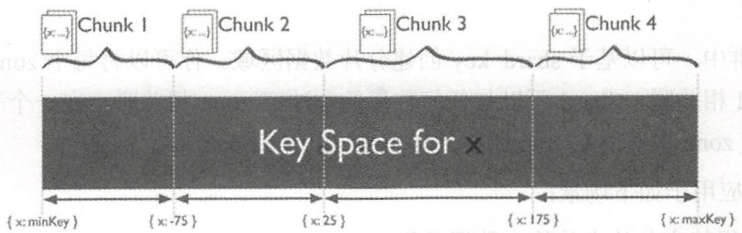
```
sh.enableSharding("<database>")
sh.shardCollection( "database.collection", { <field> : "hashed" } )
```

上述命令对 databasse 下 collection 集合的 filed 进行哈希分片。

6.17.8　范围分片

基于范围的分片是将数据划分成由 shard key 决定的连续范围，称为范围分片。在这个模型中，值比较接近的文档可能在相同的 Chunk 或 Shard 中。范围分片能够针对范围查询有着较好的效率，但是如果范围分片的范围选择得不好，会导致读写性能降低。

从下图能够看出，值相近的文档会放入一个 Chunk 中，如果查询的文档涉及的 Chunk 是连续的，则对于范围查询来说会有较好的效率。



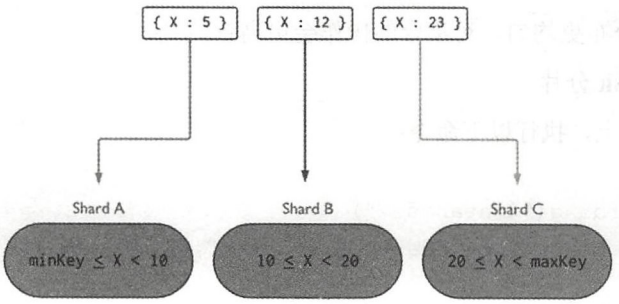
如果没有主动配置 Hash 或者 zone 区间，那么范围分片是默认的分片策略。

Shard Key 的选择

一个优秀的范围分片应该具有以下特征：

- shard key 的值范围比较大；
- 不存在高频的范围；
- 非单调递增或递减。

下图显示了使用字段 X 作为分片键的分片集群。如果 X 值范围很大、频率较低，并且以非单调速率在变化，则插入的分布可能看起来类似于以下内容。



将集合进行 Hash 分片

连接到 mongos 上，执行以下命令：

```
sh.enableSharding("<database>")
sh.shardCollection( "database.collection", { <shard key> } )
```

上述命令对 databasse 下 collection 集合的 filed 进行范围分片。

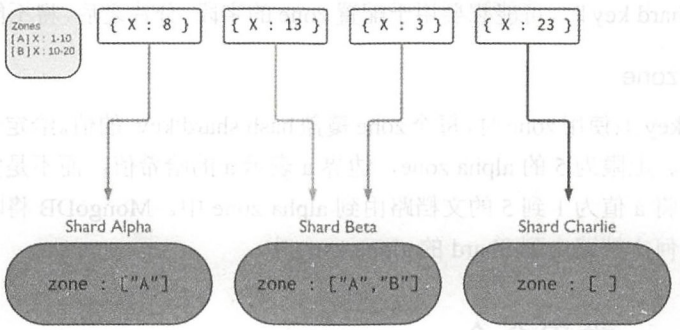
6.17.9 zone

在分片集群中，可以基于 shard key 创建分片数据区域。你可以将每个 zone 与集群中的一个或多个 Shard 相关联，Shard 可以与任何数量的非冲突 zone 相关联。在一个平衡的集群中，MongoDB 只将 zone 覆盖的 Chunk 迁移到与 zone 相关的 Shard 中。

一般 zone 应用于如下场景：

- 隔离一组特定分片上的特定数据子集。
- 确保最相关的数据位于地理位置最接近的应用程序服务器的 shard 上。
- 根据 Shard 硬件的性能将数据路由到 Shard 中。

下图显示了具有三个 Shard 和两个 zone 的分片集群。A 区表示范围，下限为 1，上限为 10。B 区表示下限为 10、上限为 20 的范围。Shard Alpha 和 Beta 有 A 区域。Shard Beta 有 A 区域也有 B 区域。Shard Charlie 没有与之相关的区域。



zone 的功能体现

每个 zone 覆盖一个或多个 shard key 值的范围，zone 覆盖的每个范围始终包含其下边界，并且不包括其上边界。简单地说就是左闭右开。

zone 不能共享范围，也不能有重叠范围。

balancer

balancer 尝试在集群中的所有分片上均匀分配分片集合的 Chunk。

对于标记为迁移的每个块，balancer 将检查每个可能的目标 Shard 以查找是否有已配置的 zone。如果 Chunk 范围符合某个 zone，则 balancer 将该 Chunk 迁移到该 Shard 对应的 zone 下，不属于任何 zone 的 Chunk 可以迁移至任何 Shard 中。

在平衡期间，如果 balancer 检测到任何 Chunk 违反给定 Shard 的配置 zone，则 balancer 将这些 Chunk 迁移至自己专属 zone 或者非冲突的 Shard 里。

在配置 zone 后，集群需要一定的时间来按照你的 zone 要求迁移这些数据，迁移的时间取决于数据量的多少。平衡完成时，读写指定 zone 中的文档只会被路由到配置了该 zone 的 Shard 里。

配置完成后，balancer 将在未来的平衡中根据 zone 的配置来进行迁移。

shard key

定义一个新的 zone 时，必须使用 shard key 中包含的字段。如果使用复合分片键，则范围必须包含分片键的前缀。

例如，给定 shard key{a：1，b：2，c：3}，创建或更新 zone 以覆盖 b 的值需要包含 a 作为

前缀，创建或更新 zone 以覆盖 c 的值需要包含 a 和 b 作为前缀。

不能使用未包含在 shard key 中的字段来创建 zone。例如，如果你想使用 zone 来根据地理位置对数据进行分区，则 shard key 将至少需要一个包含地理数据的字段。

为集合选择 shard key 时，可能想要用于配置 zone 的字段。分片之后，将不能更改 shard key。

hash key 和 zone

在 hash shard key 上使用 zone 时，每个 zone 覆盖 hash shard key 的值。给定一个 shard key{a：1}和一个下限为 1、上限为 5 的 alpha zone，边界 a 表示 a 的哈希值，而不是实际值。因此，不能保证 MongoDB 将 a 值为 1 到 5 的文档路由到 alpha zone 中。MongoDB 将哈希分片键值属于 1 到 5 范围内的任何文档路由到 Shard 的 alpha zone 中。

6.17.10 zone 常用命令

下面是 3.4 版本的新命令：sh.addShardToZone()。

```
sh.removeShardFromZone()
sh.updateZoneKeyRange()
sh.removeRangeFromZone()
```

以下是 3.4 版本 shard 查看 zone 的命令，后面的 tag 是这些命令的别名：

```
use config
db.shards.find({ tags: "NYC" })
db.tags.find({ tags: "NYC" })
```

在 shard 中添加 zone

添加 NYC zone 至 shard0000 和 shard0001，添加 SFO 和 NRT zone 至 shard0002：

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

删除 zone

删除 zone：

```
    sh.removeShardTag("shard0002", "NRT")
```

zone 添加区间

每个区间只能归属一个 zone，一个 zone 可以有多个区间：

```
    sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" },
"NYC")
    sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" },
"NYC")
    sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" },
"SFO")
    sh.addTagRange(
      "chat.messages",
      { "country" : "US", "userid" : MinKey },
      { "country" : "US", "userid" : MaxKey },
      "NA"
    )
```

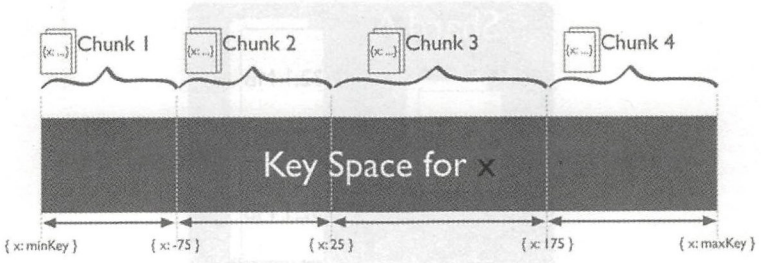
4. 删除 zone 区间

3.4 版本新特性，即删除 zone 的部分区间范围：

```
    sh.removeRangeFromZone("records.user",  {zipcode:  "10001"},  {zipcode:
"10281"})
```

6.17.11　Chunk

如下图所示，MongoDB 使用 shard key 将数据分成若干个 Chunk，Chunk 是一个集合数据中的子集。每个 Chunk 具有基于 shard key 的范围，左闭右开。



mongos 根据 shard key 的值来写入对应的 Chunk，当 MongoDB 超出配置的 Chunk 大小时，会分割 Chunk。插入和更新都可以触发 Chunk 拆分。

Chunk 可以表示的最小范围是单个唯一的 shard key 值，只包含具有单个 shard key 值文档的 Chunk 不能被分割。

Chunk 的大小

MongoDB 中的默认块大小是 64MB，可以增加或减小 Chunk 大小。

小 Chunk 可以更加均匀地分布数据，但会导致更频繁的迁移，这在查询路由（mongos）层会有一定开销。

大 Chunk 触发的迁移更少，但也会导致数据分布不均匀。

Chunk 大小会影响每个要迁移的 Chunk 的最大文档数。

为一个已经存在的集合做分片操作时，Chunk 的大小会影响该集合的最大大小值。Chunk 分割后，Chunk 大小不会限制集合大小。

限制

更改 Chunk 大小会影响 Chunk 的拆分，所以会受到一些限制。

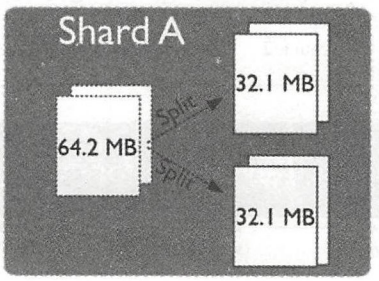
自动分割仅在插入或更新期间发生，如果降低 Chunk 大小，则可能需要一段时间才能将所有 Chunk 分割完。

拆分不能被“撤消”，如果增加 Chunk 大小，现有的 Chunk 必须通过插入或更新来增长，直至达到阈值后才能拆分。

Chunk Split

分裂是为了 Chunk 不能太大，当一个 Chunk 增长到指定大小后，或者如果该 Chunk 中的文档数量超过了每个 Chunk 的最大文档数时，MongoDB 将根据该 Chunk 的 shard key 值来分割 Chunk。插入和更新可能触发拆分。

如下图所示，达到默认的 64MB 大小后会被分裂。



分裂可能会导致 Chunk 在 Shard 集群中分布不均匀，这时候 balancer 会平衡各 Shard 中 Chunk 的数量。

Chunk 迁移

MongoDB 迁移 Chunk 的情况有：

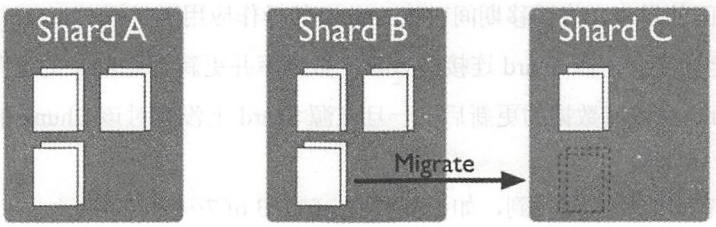
手动迁移，一般我们不需要手动迁移 Chunk。

自动迁移，balancer 进程在各个 Shard 中分布不均匀时进行 Chunk 迁移。

Balancing

balancer 是管理 Chunk 迁移的后台进程，如果各 Shard 之间的 Chunk 数量差值超过了阈值，则 balancer 开始在集群中迁移 Chunk 以确保数据的均匀分布。

如下图所示，当达到了阈值时，balancer 会开始迁移 Chunk。



迁移阈值如下表所示。

Chunks 数量	迁 移 阈 值
小于 20	2
20~79	4
大于 80	8

可以在想要的 Shard 中使用 zone 来配置固定 Chunk。

不可分割的 Chunk

在某些情况下，Chunk 可能超出指定的块大小，但不能进行拆分。最常见的情况是，块代表单个分片键值。由于块不能分割，所以它会继续增长，超出块的大小，成为一个巨大的块。这些巨型块可能会成为一个性能瓶颈，因为它们继续增长，尤其是当分片键值高频率出现的时候很容易遇到瓶颈。

新数据或新碎片的添加可能导致集群内的数据分布不均衡。一个特定的分片可能比另一个分片获得更多的块，或者块的大小可能超过配置的最大块大小。

6.17.12 Chunk 迁移

整个集群只能有一个 mongos 的 balancer 迁移，一次只迁移一个块（3.4 版本有变更，$n/2$ n=shard num），采用锁来实现。

balancer 运行在 config server 的主节点上，从 3.4 版本起，Chunk 的迁移分为 7 个步骤：

（1）balancer 进程将 moveChunk 命令发送到源 Shard 中。

（2）源 Shard 使用内部 moveChunk 命令开始移动。在迁移过程中，对该 Chunk 的操作依旧在源 Shard 上进行。源 Shard 依旧负责该 Chunk 的写入操作。

（3）目标 Shard 开始创建所需索引。

（4）目标 Shard 开始请求 Chunk 中的文档并开始接收数据的复制。

（5）在接收完待迁移 Chunk 的最后一个文档后，目标 Shard 将启动一个同步进程，这个进程会拉取迁移期间的日志，将迁移期间对该 Chunk 的操作应用至目标 Chunk 中。

（6）当完全同步时，源 Shard 连接到 config 数据库并更新新 Chunk 的位置元数据。

（7）源 Shard 完成元数据的更新后，一旦在源 Shard 上没有对该 Chunk 的操作，源 Shard 会默认异步删除 Chunk。

这 7 个步骤在日志中也能找到，如 1 of 7，2 of 7，3 of 7…

```
    2018-01-04T06:53:06.865+0000 I SHARDING [conn102845] about to log metadata
event into changelog: { _id: "ip-10-41-58-141-2016-02-15T06:53:06.
865+0000-56c175d207f175cfb9224558", server: "ip-10-41-58-141", clientAddr:
"192.168.1.248:48224", time: new Date(1455519186865), what: "moveChunk.from",
ns: "mydomain.Sessions", details: { min: { a: ObjectId
('5334b6f2645cff3b5097f4f9'), _id: ObjectId('55f705bca21c0f117ccc613c') }, max:
{ a: ObjectId('5334b6f2645cff3b5097f4f9'), _id: ObjectId
('55fb9231584ad132f5207391') }, step 1 of 7: 0, step 2 of 7: 310, step 3 of 7:
15, step 4 of 7: 8, step 5 of 7: 299, step 6 of 7: 1,step 7 of 7: 0, to: "shmydomain3",
from: "shmydomain1", note: "success" } }
    2018-01-04T06:53:06.993+0000 I SHARDING [conn102845] distributed lock
'mydomain.Sessions/ip-10-41-58-141:27017:1455127062:862901621' unlocked.
```

使用 3.0 版本及以前版本的 Sharded cluster 可能会遇到一个问题，即停止写入数据后，数据目录里的磁盘空间占用还会一直增加。

上述行为是由 sharding.archiveMovedChunks 的配置项决定的，该配置项在 3.0 版本及以前的版本中默认为 true，即在 move chunk 时，源 Shard 会将迁移的 Chunk 数据归档在数据目录里，

当出现问题时可用于恢复。也就是说，Chunk 发生迁移时，源节点上的空间并没有释放出来，而目标节点又占用了新的空间。

迁移后立即删除原 Chunk

数据迁移完后，源 Shard 需要将迁移完的 Chunk 移除。默认情况下，源 Shard 会将删除 Chunk 的任务加到一个后台队列中，在后台异步删除，然后 balancer 就可以启动下一次的 Chunk 迁移。用户可以设置_waitForDelete 为 true(默认为 false)，让源 Shard 在 Chunk 迁移完后同步删除 Chunk 数据。

```
>use config
>db.settings.update(
>   { "_id" : "balancer" },
>   { $set : { "_waitForDelete" : true } },
>   { upsert : true }
>)
```

Chunk 迁移场景

场景 1：

当多个 Shard 上 Chunk 数量分布不均时，MongoDB 会自动在 Shard 间迁移 Chunk，尽可能让各个 Shard 上 Chunk 数量均匀分布，就是大家经常说到的负载均衡。

场景 2：

用户调用 removeShard 命令后，被移除 Shard 上的 Chunk 就需要被迁移到其他的 Shard 上，等该 Shard 上没有数据后安全下线(注意：Shard 上没有分片的集合，需要手动进行 movePrimary 来迁移，系统不会自动迁移)。

场景 3：

MongoDB sharding 支持 shard tag 功能，可以对 Shard 及 shard key range 打标签，系统会自动将对应 range 的数据迁移到拥有相同 tag 的 Shard 上。例如：

```
mongos> sh.addShardTag("shard-hz", "hangzhou")
mongos> sh.addShardTag("shard-sh", "shanghai")
mongos> sh.addTagRange("shtest.coll", {x: 1}, {x: 1000}, "hangzhou")
mongos> sh.addTagRange("shtest.coll", {x: 2000}, {x: 5000}, "shanghai")
```

以上命令对 2 个 Shard 添加了标签，对某个集合的 shard key range 也添加了标签，这样该集合里 x 值为[1, 1000)的文档都会分布到 shard-hz 上，而 x 值为[2000, 5000)的文档则会分布到

shard-sh 里。

6.17.13 chunksize

MongoDB 默认的 chunkSize 为 64MB，文档数量不能超过 250 000 个，如无特殊需求，建议保持默认值；chunkSize 会直接影响 Chunk 分裂、迁移等行为。

修改 chunksize 默认大小：

```
>use config
>db.settings.save( { _id:"chunksize", value: 100 } )
```

chunkSize 越小，Chunk 分裂及迁移越多，数据分布越均衡，频繁迁移会增加 mongos 的查询开销；

chunkSize 太小，容易出现 jumbo chunk（即 shardKey 的某个取值出现频率很高，这些文档只能放到一个 Chunk 里，无法再分裂）而无法迁移；

chunkSize 越大，Chunk 分裂及迁移会越少，但可能导致数据分布不均；

chunkSize 越大，则可能出现 Chunk 内文档数太多（Chunk 内文档数不能超过 25 万个）而无法迁移。

Chunk 自动分裂只会在数据写入（insert/update）时触发，所以如果将 chunkSize 改小，系统需要一定时间来将 Chunk 分裂到指定大小。

Chunk 只会分裂、不会合并，所以即使将 chunkSize 改大，现有的 Chunk 数量不会减少，但 Chunk 大小会随着写入不断增长，直到达到目标大小为止。

jumbo chunk

MongoDB 默认的 chunk size 为 64MB，如果 Chunk 超过 64MB 并且不能分裂（比如所有文档的 shard key 都相同），则会被标记为 jumbo chunk，Balancer 不会迁移这样的 Chunk，从而可能导致负载不均衡，应尽量避免。

一旦出现了 jumbo chunk，如果对负载均衡要求不高，则不去关注也没什么影响，因为并不会影响到数据的读写访问。如果一定要处理，可以尝试如下方法：

对 jumbo chunk 进行 split 操作，一旦 split 成功，mongos 会自动清除 jumbo 标记。

对于不可再分的 Chunk，如果该 Chunk 已不再是 jumbo chunk，可以尝试手动清除 Chunk 的 jumbo 标记（注意先备份 config 数据库，以免误操作导致 config 库损坏）。

最后的办法，调大 chunk size，当 Chunk 大小不再超过 chunk size 时，jumbo 标记最终会被

清理，但这是治标不治本的方法，随着数据的写入仍然会再出现 jumbo chunk，根本的解决办法还是合理地规划 shard key。

chunksize 与分片集合最大大小的关系图如下所示。

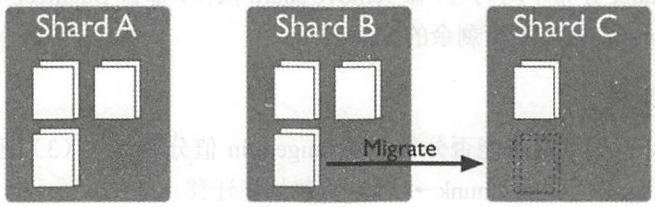
Average Size of Shard Key Values	512 bytes	256 bytes	128 bytes	64 bytes
Maximum Number of Splits	32,768	65,536	131,072	262,144
Max Collection Size (64 MB Chunk Size)	1 TB	2 TB	4 TB	8 TB
Max Collection Size (128 MB Chunk Size)	2 TB	4 TB	8 TB	16 TB
Max Collection Size (256 MB Chunk Size)	4 TB	8 TB	16 TB	32 TB

6.17.14　Balancer

MongoDB Balancer 是一个后台进程，用于监视每个 Shard 上 Chunk 的数量。当给定 Shard 上 Chunk 的数量达到特定的迁移阈值时，Balancer 尝试在 Shard 之间自动迁移 Chunk，最终让每个 Shard 具备相同数量的 Chunk。

Shard 集群的平衡过程对于用户和应用层是完全透明的，但是在迁移过程中会对性能产生影响。

如下图所示，当达到阈值时，Balancer 会开始迁移 Chunk。



集群 Balancer

Balancer 进程负责将 Shard 集合的 Chunk 均匀地重新分配给每个 Shard。默认情况下，Balancer 进程始终处于启用状态。

在 3.4 版本中，Balancer 运行在 config server 的主节点上，而在早期版本中 Balancer 是运行在 mongos 上的。当 Balancer 进程处于活动状态时，config server 副本集的主服务器通过修改

config 数据库的 lock 集合中的文档来获取“平衡器锁”，这个“平衡器锁”只能自己主动释放。

为了解决分片集合中 Chunk 分布不均匀的问题，Balancer 将具有较多 Chunk 的 Shard 迁移至具有较少 Chunk 的 Shard 中，Balancer 迁移 Chunk 这个过程会一直持续到在整个 Shard 中存在 Chunk 的均匀分布为止。

Chunk 迁移在带宽和工作负载方面会带来一些开销，这两者都会影响数据库的性能。Balancer 试图通过以下方式将影响降至最低：

（1）在任何给定的时间内最多限制一次迁移，即 Shard 不能同时参与多个 Chunk 迁移。如果要从一个 Shard 中迁移多个 Chunk，那么 Balancer 一次只迁移一个 Chunk。但这个特性在 3.4 版本中做了更改提升，从 MongoDB 3.4 开始，MongoDB 可以执行并行 Chunk 迁移。对于具有 n 个 Shard 的分片集群，MongoDB 最多可以执行 $n/2$ 个 Chunk 同时迁移。

（2）只有在分片集合中 Chunk 数量最多的 Shard 与 Chunk 数量最少的 Shard 差值达到迁移阈值时，才开始迁移。

（3）可以关闭 Balancer 来进行临时维护。

（4）可以配置 Window，即限制只有在固定的时间段内可以开始迁移操作。

集群中增加或移除 Shard

由于新 Shard 里没有 Chunk，所以向集群添加 Shard 会产生不平衡。当 MongoDB 开始立即将数据迁移到新的 Shard，这个操作会需要一定的时间才能完成。

从群集中删除 Shard 会产生类似的不平衡，因为驻留在该分片上的 Chunk 必须在整个群集中重新分发。当 MongoDB 删除某个 Shard 的时候，Shard 会出现 draining 状态，这表示该 Shard 正在被删除，在集群平衡之前可能需要一些时间。在此过程中，不要关闭服务器。

当集群中的 Chunk 分布不均匀时，做 remove shard 操作，那么 Balancer 将首先移除 draining 状态 Shard 里的 Chunk，然后平衡剩余的 Shard。

Balancer 迁移流程

（1）获取信息→（2）检查能否分裂（以 range.min 值分裂）→（3）迁移 draining 状态的 Shard→（4）迁移 tag 不匹配的 Chunk→（5）负载均衡迁移。

Balancer 迁移阈值如下表所示。

集合的 chunk 数量	迁 移 阈 值
< 20	2
20~79	4
>=80	8

如果该集合的任何两个分片上块数目之间的差异小于 2，或者块迁移失败，则平衡器将停止在目标集合上运行。

异步 Chunk 迁移清理

要从一个分片中迁移多个 Chunk，Balancer 一次只迁移一个 Chunk。但是在开始下一个 Chunk 迁移之前，Balancer 不会等待当前迁移的删除阶段完成。

这种排队行为允许 Shard 在严重不平衡的集群情况下更迅速地卸载 Chunk，例如当执行初始数据加载而没有预分割和添加新 Shard 时。

在某些情况下，删除阶段可能会持续更长时间。如果多个删除处在 queue 排队阶段，那么可以在副本集的主库上删除孤儿文档。

_waitForDelete（可用作 Balancer 的设置及 moveChunk 命令）可以改变行为，会在删除完过期 Chunk 后再进行下一个 Chunk 迁移。_waitForDelete 通常用于内部测试，不能依赖。

Chunk 的迁移和复制

在 Chunk 迁移期间，_secondaryThrottle 值决定 Balancer 何时继续处理 Chunk 中的下一个文档：如果设置为 true，那么在默认情况下，Chunk 迁移期间的每个文档移动将在 Balancer 继续处理下一个文档之前传播到至少一个 Secondary 上，相当于{w：2}的 write Concern。

注意

Balancer 配置文档中的 writeConcern 字段允许使用_secondaryThrottle 选项指定不同的 writeConcern。如果为 false，则 Balancer 不会等待复制到 Secondary 节点，而是继续下一个文档。

在 Chunk 迁移完成但还没有更新 config 库元数据信息前，MongoDB 会短暂地暂停所有应用程序对正在迁移 Chunk 的集合进行读写。

如果 Chunk 中文档的数量超过 25 万个，则 MongoDB 不能移动 Chunk。

6.17.15　Balancer 运维

```
    >sh.getBalancerState()、sh.isBalancerRunning()：获取 sharding 集群 Balance 是否开启；
    >sh.stopBalancer()：停止 Balancer;
    >sh.startBalancer()：开启 Balancer;
    >sh.disableBalancing("students.grades")：针对 students 库 grades 集合关闭 Balance;
    >sh.enableBalancing("students.grades")；针对 students 库 grades 集合开启 Balance。
```

配置窗口避免白天操作影响业务，只在凌晨 2 点到 6 点工作：

```
>use config
>db.settings.update(
>   { _id: "balancer" },
>   { $set: { activeWindow : { start : "02", stop : "06" } } },
>   { upsert: true }
>)
```

2 个参数需要组合起来使用，意思是如果_secondaryThrottle 为 true，则使用 writeConcern 选项来指定迁移时写数据的策略；如果_secondaryThrottle 为 false，则使用{w: 1}，如下命令将 writeConcern 设置为{w: majority}。

如果没有设置，则默认使用{w: 2}，要求至少写到目标 2 个节点（若目标 shard 是单节点，则退化为{w: 1}）：

```
>use config
>db.settings.update(
>   { "_id" : "balancer" },
>   { $set : { "_secondaryThrottle" : true ,
>              "writeConcern": { "w": "majority" } } },
>   { upsert : true }
>)
```

从 3.4 版本起，对于 WT 存储引擎，默认_secondaryThrottle 为 false，对于 MMAPv1 存储引擎，该值依旧默认是 true。

数据迁移完后，源 Shard 需要将迁移完的 Chunk 进行移除，默认情况下，源 Shard 会将删除 Chunk 的任务加到一个后台队列，在后台异步删除，然后 Balancer 就可以启动下一次的 Chunk 迁移。用户可以设置_waitForDelete 为 true（默认为 false），让源 Shard 在 Chunk 迁移完后同步删除 Chunk 数据。

```
>use config
>db.settings.update(
>   { "_id" : "balancer" },
>   { $set : { "_waitForDelete" : true } },
>   { upsert : true }
>)
```

MognoDB sharding 默认 chunkSize 为 64MB，默认设置在绝大多数场景都是合适的。在某些场景下，用户可能需要修改 chunkSize 配置。

如下命令将 chunkSize 修改为 100MB：

```
>use config
>db.settings.save( { _id:"chunksize", value: 100 } )
```

6.18 Troubleshoot Sharded Clusters

某个应用服务器或者 Mongos 宕机

如果每个应用程序服务器都有自己的 Mongos 实例，则其他应用程序服务器可以继续访问数据库。

Mongos 实例不保持持久状态，它们可以重新启动（启动过程中为 Unavailable），且不会丢失任何状态或数据。

当一个 Mongos 实例启动时，它会从 config server 获取数据，并开始路由查询。

在 Sharding 集群中，其中一个 Mongod 进程宕机

副本集本身提供非常好的高可用性能。

如果宕机的是主库，则副本集会选出一个新的主库。

如果宕机的是从库，则会断开与主库的连接，继续保留所有数据。

在 3 个成员副本集中，即使单个成员遇到灾难性故障，另外两个成员也有完整的数据副本。

经常检查可用性，如果系统不可恢复，则应尽快替换掉出问题的服务器，将一台新的成员加入到副本集中。

某个 Sharding 中的所有成员都是 Unavailable 状态

如果某个分片的所有节点都不可用，则该分片中的所有数据都不可用。但是其他分片上的数据将保持可用，并且可以继续将数据读取和写入非故障分片。

另外，应用程序必须能够处理返回的部分结果集，此时 DBA 应该调查中断的原因，并尝试尽快恢复分片。

Configserver 副本集成员不可用

从 MongoDB 3.2 开始，分片集群的配置服务器可以部署为副本集，同时必须运行 WiredTiger 存储引擎。

MongoDB 3.2 不推荐使用独立的三个镜像 Mongod 实例作为配置服务器。

副本集为配置服务器提供高可用性：

（1）如果主节点不可用，副本集将选择新的主服务器。

（2）如果副本集配置服务器丢失其主节点，且无法选择主节点，则集群的元数据将变为只读。仍然可以从分片读取和写入数据，但是在 Primary 可用之前无法进行块迁移或块分割。如果所有配置数据库不可用，则集群瘫痪。

Configserver 数据过旧导致游标失败

当一个或多个 Mongos 实例尚未从配置数据库更新其元数据的缓存时，查询会返回以下警告：

```
could not initialize cursor across all shards because : stale config detected
```

该警告会重复出现直到所有的 Mongos 实例刷新其缓存。

要强制实例刷新其缓存，执行 flushRouterConfig 命令。

分片键与集群可用性

选择分片键时最重要的考虑因素是：

（1）确保 MongoDB 能够在分片间均匀分配数据。

（2）写操作可以遍布整个集群。

（3）确保 Mongos 可以将大多数查询发送到指定的 Mongod。

更多：

（1）每个分片应该是副本集，如果某个 Mongod 实例不可用，则副本集成员将选择另一个实例作为主节点并继续提供服务。如果整个分片由于某种原因无法访问，则该数据将不可用。

（2）如果分片键允许 Mongos 将大多数操作路由到单个分片上，则某个单分片的故障只会导致部分数据不可用。

（3）如果分片键的分布要求每个操作需要操作所有分片，任意一个分片不可用将导致整个集群不可用。

这也说明选择合适的分片键对于路由查询到单个分片的重要性。

config server 连接串错误

从 MongoDB 3.2 开始，config server 可以部署为副本集。

mongos 节点配置文件必须指定相同的 config server 的副本集名称，但可以指定副本集中不

同成员的主机名和端口。

从 3.4 版本开始，不再支持使用 3 个独立节点的镜像 mongod 实例作为配置服务器（SCCC）。

将分片集群升级到 3.4 版本之前，必须将配置服务器从 SCCC 转换为 CSRS。

对于早期版本的 MongoDB 分片集群，configserver 使用 3 个镜像 mongod 实例，分片集群中的每个 mongos 实例必须指定相同的 configDB 字符串。

迁移 config server 时避免停机

使用 CNAME 来识别配置服务器，可以重新命名和重新编号配置服务器。

Move Chunk 报错

Chunk migration 结束时，分片必须连接到 config server 才能更新集群元数据中的块记录。

如果分片无法连接到 config server，MongoDB 会报告以下错误：

```
ERROR: moveChunk commit failed: version is at <n>|<nn> instead of
<N>|<NN>" and "ERROR: TERMINATING"
```

当发生这种情况时，副本集的 Primary 节点停止 move 操作以保护数据的一致性。

如果 Secondary 成员可以访问配置数据库，则在选举新主之后，分片上的数据可以再次被访问。

6.19　在线开启认证

环境

192.168.1.248:28000（主）

192.168.1.249 （从）

192.168.1.251 （从）

目标：将 no auth 副本集集群实现 0 down time 开启认证。

操作步骤

（1）创建角色和用户。

（2）客户端调整登录方式为使用用户名密码（在完成本教程时，副本集会拒绝未经身份验证的客户端连接。现在执行此步骤可确保客户端在转换之前和之后连接到副本集）。

（3）创建 keyfile 并复制到每个副本集成员机器上。

（4）副本集从库成员逐个开启（如果只开启 keyFile 那么成员之间及客户端连接必须经过认证，而其他成员还没有开启 keyfile，因此会出现这个集群的状态是 other 和 recovering）：

```
security:
  keyFile: <path-to-keyfile>
  transitionToAuth: true
  #clusterAuthMode: "keyFile"这个默认就是 keyFile,加不加该参数都可以。
```

（5）主库 stepdown 并开启上述第 4 步(这一步完成后整个集群都开启了 transitionToAuth+keyfile，此时使用或者不使用密码登录都可以操作库)。

（6）副本集从库成员逐个去掉 transitionToAuth（去掉后就必须使用密码登录而没去掉的成员依旧能够使用非密码登录）。

（7）主库 stepdown 并关闭上述第 4 步的 transitionToAuth。

相关日志

```
###可以看到在当前没有配置 keyfile 时不使用用户登录依旧能够操作数据库###:
[root@HE1 ~]# /home/work/mongodb/bin/mongo --host 127.0.0.1 --port 28000
heleitest:PRIMARY> db.helei.insert({noauth:2})
WriteResult({ "nInserted" : 1 })
heleitest:PRIMARY> db.helei.find()
{ "_id" : ObjectId("5a1778cc4292190aafbe6965"), "a" : 1 }
{ "_id" : ObjectId("5a17793aa30fc0968a65cbe4"), "noauth" : 2 }
##########################################################################
 ############使用认证登录也可以操作########
[root@HE1 ~]# /home/work/mongodb/bin/mongo 127.0.0.1:28000/helei -u
sys_admin -p MANAGER --authenticationDatabase admin
heleitest:PRIMARY> db.helei.insert({auth:3})
heleitest:PRIMARY> db.helei.find()
{ "_id" : ObjectId("5a1778cc4292190aafbe6965"), "a" : 1 }
{ "_id" : ObjectId("5a17793aa30fc0968a65cbe4"), "noauth" : 2 }
{ "_id" : ObjectId("5a1779db7628083bf4ebd03f"), "auth" : 3 }
################################
###升级所有客户端，将其配制成认证模式，以适应后续开启认证###:
/home/work/mongodb/bin/mongo --host heleitest/192.168.1.248:28000,192.168.
1.249:28000,192.168.1.251:28000 -u helei_wr -p MANAGER --authenticationDatabase
admin
```

这个时候使用错误的密码是不会登录的：

```
    /home/work/mongodb/bin/mongo                                    --host
heleitest/192.168.1.248:28000,192.168.1.249:28000,192.168.1.251:28000     -u
helei_wr -p MANAGER1 --authenticationDatabase admin
    2017-11-23T18:15:21.979-0800 E QUERY    [thread1] Error: Authentication
failed. :
    ############################################################
    ############如果不开启 transitionToAuth 只开启 keyfile############
    heleitest:OTHER>
    heleitest:RECOVERING>
    2017-11-23T19:02:42.582-0800 I ACCESS      [conn1] Unauthorized: not
authorized on admin to execute command { replSetHeartbeat: "heleitest",
configVersion: 1, from: "192.168.1.248:28000", fromId: 0, term: 1 }
    2017-11-23T19:02:43.416-0800                I                ASIO
[NetworkInterfaceASIO-Replication-0] Failed to connect to 192.168.1.248:28000
- AuthenticationFailed: Authentication failed.
    2017-11-23T19:02:43.416-0800                I                ASIO
[NetworkInterfaceASIO-Replication-0] Dropping all pooled connections to
192.168.1.248:28000 due to failed operation on a connection
    2017-11-23T19:02:43.416-0800 I REPL     [ReplicationExecutor] Error in
heartbeat request to 192.168.1.248:28000; AuthenticationFailed: Authentication
failed.
    ###########开启 transitionToAuth 和 keyfile###########
```

这时候由于只有这个成员开启了 keyfile 认证，并且也开启了 transitionToAuth，使得其可以与其他成员通信，并且使用密码和不使用密码都是可以登录的，但错误的密码是不能登录的。

```
    [root@HE4 ~]# /home/work/mongodb/bin/mongo --host 127.0.0.1 --port 28000
    heleitest:SECONDARY>
    [root@HE4  ~]#  /home/work/mongodb/bin/mongo  127.0.0.1:28000/helei  -u
sys_admin -p MANAGER --authenticationDatabase admin
    heleitest:SECONDARY>
    [root@HE4  ~]#  /home/work/mongodb/bin/mongo  127.0.0.1:28000/helei  -u
sys_admin -p MANAGER1 --authenticationDatabase admin
    2017-11-23T19:08:42.884-0800 E QUERY    [thread1] Error: Authentication
failed. :
```

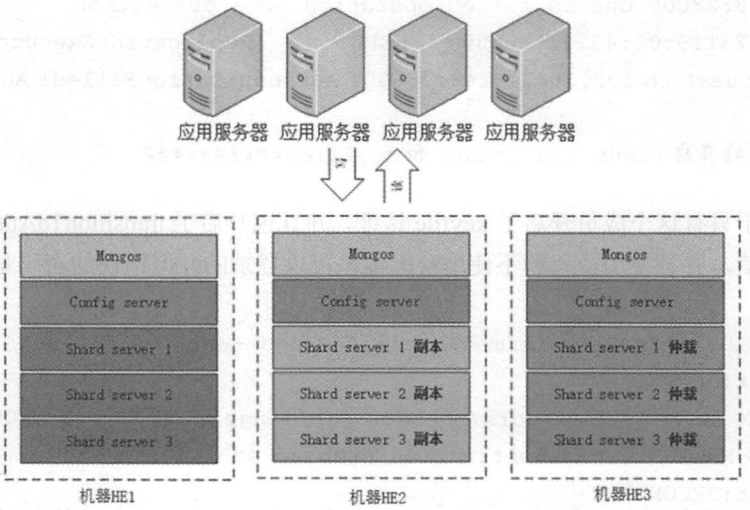
日志信息：

```
    2017-11-23T19:05:46.123-0800 I ACCESS   [NetworkInterfaceASIO-RS-0] Failed
to authenticate in transitionToAuth, falling back to no authentication.
    2017-11-23T19:05:46.123-0800  I  ASIO         [NetworkInterfaceASIO-RS-0]
Successfully connected to 192.168.1.249:28000, took 13ms (3 connections now open
to 192.168.1.249:28000)


    ####下面这个在替换最后一个从即原主库的时候不会再出现，因为整个集群都配置了
transitionToAuth#########
    2017-11-23T19:17:33.250-0800 I ACCESS   [NetworkInterfaceASIO-RS-0] Failed
to authenticate in transitionToAuth, falling back to no authentication.
```

6.20 分片架构搭建

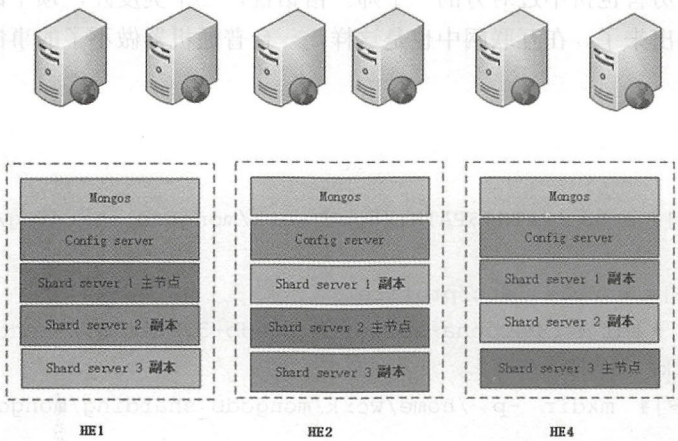
架构设计

如果用 3 台机器来搭建分片架构，则一个分片架构大致如下图所示。



从上述架构中你看出什么端倪了吗？没错，当机器出现故障的时候，会直接导致整个分片不可用，例如机器 HE1 出现问题，则 Shard server 1 数据不可用，对整个集群而言，这不是我们所希望的。

再来看一下架构，如下图所示。



环境

```
mongos:27000
config server:27001
shard1:27017
shard2:27018
shard3:27019
```

在 mongod 和 config 副本集配置中，端口号不同、路径不同、副本集名不同及 clusterRole 不同。mongos 的 sharding 配置的是 configDB。

应用配置多个 mongos，实现负载均衡和高可用，避免单节点 mongos 故障带来整个集群不可用的隐患。

mongos，数据库集群请求的入口，所有的请求都通过 mongos 进行协调，不需要在应用程序添加一个路由选择器，mongos 自己就是一个请求分发中心，它负责把对应的数据请求转发到对应的 Shard 服务器上。在生产环境中通常有多个 mongos 作为请求的入口，防止当其中一个挂掉时所有的 MongoDB 请求都没有办法操作。

config server，顾名思义为配置服务器，存储所有数据库元信息（路由、分片）的配置。mongos 本身没有物理存储分片服务器和数据路由信息，只是缓存在内存里，配置服务器则实际存储这些数据。mongos 第一次启动或者关掉重启就会从 config server 加载配置信息，以后如果配置服务器有信息变化会通知所有的 mongos 更新自己的状态，这样 mongos 就能继续准确路由。在生产环境通常有多个 config server 配置服务器，因为它存储了分片路由的元数据，这个一定不要丢失！就算挂掉其中一台，只要还有存货，MongoDB 集群就不会挂掉。

Shard，这就是传说中的分片了。上面提到一个机器就算能力再大也有天花板，就像军队打仗一样，一个人再厉害也拼不过对方的一个师。俗话说，三个臭皮匠，顶个诸葛亮，这个时候团队的力量就凸显出来了。在互联网中也是这样，一台普通机器做不了的事情可以用多台机器来做。

搭建过程

```
    [root@HE1 ~]# echo 'PATH=$PATH:/home/work/mongodb_sharding/3.2.12/bin' >>
/etc/profile
    [root@HE1 ~]# source /etc/profile
    [root@HE1 ~]# tar xvf percona-server-mongodb-3.2.12-3.2-centos6-x86_64.tar.
gz -C /home/work/
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27017/data/
db_27017
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27017/etc
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27017/log
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27017/tmp
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27018
/data/db_27018
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27018/etc
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27018/log
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27018/tmp
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_ 27019/data/
db_27019
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27019/etc
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27019/log
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongodb_27019/tmp
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongos_27000/data/
db_27000
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongos_27000/etc
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongos_27000/log
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/mongos_27000/tmp
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/config_27001/data/
db_27001
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/config_27001/etc
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/config_27001/log
    [root@HE1 ~]# mkdir -p /home/work/mongodb_sharding/config_27001/tmp
    [root@HE1 ~]# mv /home/work/percona-server-mongodb-3.2.12-3.2/bin /home/
```

```
work/mongodb_sharding/3.2.12/
    [root@HE1 ~]#
    [root@HE1 ~]#
    [root@HE1 ~]# openssl rand -base64 756 > /home/work/mongodb_sharding
/mongodb_27017/etc/mongodb-keyfile-benchmark
    [root@HE1 ~]# chmod 400 /home/work/mongodb_sharding/mongodb _27017/
etc/mongodb-keyfile-benchmark
    [root@HE1 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/mongodb
-keyfile-benchmark /home/work/mongodb_sharding/mongodb_27018/etc/
    [root@HE1 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/
etc/mongodb-keyfile-benchmark /home/work/mongodb_sharding/mongodb_27019/etc/
    [root@HE1 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/mongodb
-keyfile-benchmark /home/work/mongodb_sharding/config_27001/etc/
    [root@HE1 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/
mongodb-keyfile-benchmark /home/work/mongodb_sharding/mongos_27000/etc/
    [root@HE1 ~]# chown -R work. /home/work/mongodb_sharding
```

依次复制到各个机器上：

```
    [root@HE1 ~]#scp -rp /home/work/mongodb_sharding/mongodb_27017/etc/
mongodb-keyfile-benchmark
work@HE2:/home/work/mongodb_sharding/mongodb_27017/etc/
    [root@HE1 ~]#scp -rp /home/work/mongodb_sharding/mongodb
_27017/etc/mongodb-keyfile-benchmark
work@HE4:/home/work/mongodb_sharding/mongodb_27017/etc/
```

在 HE2 和 HE4 机器上执行如下 CP 动作，让 key 文件复制完全：

```
    [root@HE2 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/mongodb
-keyfile-benchmark /home/work/mongodb_sharding/mongodb_27018/etc/
    [root@HE2 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/mongodb
-keyfile-benchmark /home/work/mongodb_sharding/mongodb_27019/etc/
    [root@HE2 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc
/mongodb-keyfile-benchmark /home/work/mongodb_sharding/config_27001/etc/
    [root@HE2 ~]# cp -rp /home/work/mongodb_sharding/mongodb_27017/etc/
mongodb-keyfile-benchmark /home/work/mongodb_sharding/mongos_27000/etc/
    [root@HE2 ~]# chown -R work. /home/work/mongodb_sharding
```

每一个 mongod 进程的 etc 文件要变更端口号、路径和副本集名：

```
    [root@HE1  ~]#  vi  /home/work/mongodb_sharding/mongodb_27017/etc/mongodb
_27017.conf
    systemLog:
      destination: file
      path: "/home/work/mongodb_sharding/mongodb_27017/log/mongodb_27017.log"
      logAppend: true

    #net Options
    net:
      port: 27017
      maxIncomingConnections: 10240
      http:
         enabled: false
         JSONPEnabled: false
         RESTInterfaceEnabled: false

    #security Options
    security:
      authorization: 'enabled'
      keyFile:
/home/work/mongodb_sharding/mongodb_27017/etc/mongodb-keyfile-benchmark
      clusterAuthMode: "keyFile"

    #storage Options
    storage:
      engine: "wiredTiger"
      dbPath: /home/work/mongodb_sharding/mongodb_27017/data/db_27017
      indexBuildRetry: true
      journal:
        enabled: true
        commitIntervalMs: 100
      wiredTiger:
        engineConfig:
          cacheSizeGB: 1 #根据内存大小一般给 50%~70%内存
          journalCompressor: "snappy"
          directoryForIndexes: true
```

```
    collectionConfig:
      blockCompressor: "snappy"
    indexConfig:
      prefixCompression: true

sharding:
  clusterRole: shardsvr

#replication Options
replication:
   oplogSizeMB: 40960 #40GB
   replSetName: shard1 #副本集名称

#operationProfiling Options
operationProfiling:
  slowOpThresholdMs: 500
  mode: "slowOp"

processManagement:

  fork: true
```

在 HE1、HE2、HE4 上启动 mongod 进程：

```
[root@HE1 ~]# su - work
[work@HE1 ~]$ numactl --interleave=all /home/work/mongodb_sharding/3.2.12/
bin/mongod  -f  /home/work/mongodb_sharding/mongodb_27017/etc/mongodb_27017.
conf
[work@HE1          ~]$          numactl          --interleave=all
/home/work/mongodb_sharding/3.2.12/bin/mongod                          -f
/home/work/mongodb_sharding/mongodb_27018/etc/mongodb_27018.conf
[work@HE1          ~]$          numactl          --interleave=all
/home/work/mongodb_sharding/3.2.12/bin/mongod -f /home/work/mongodb_sharding/
mongodb_27019/etc/mongodb_27019.conf
```

config 文件和副本集的区别是 clusterRole: configsvr 和 replSetName: configserver，同时修改路径和端口号：

```
[work@HE1    ~]$    vi    /home/work/mongodb_sharding/config_27001/etc/
```

```
mongodb.conf
    systemLog:
      destination: file
      path: "/home/work/mongodb_sharding/config_27001/log/config_27001.log"
      logAppend: true

    #net Options
    net:
      port: 27001
      maxIncomingConnections: 10240
      http:
          enabled: false
          JSONPEnabled: false
          RESTInterfaceEnabled: false

    #security Options
    security:
      authorization: 'enabled'
      keyFile:
/home/work/mongodb_sharding/config_27001/etc/mongodb-keyfile-benchmark
      clusterAuthMode: "keyFile"

    #storage Options
    storage:
      engine: "wiredTiger"
      dbPath: /home/work/mongodb_sharding/config_27001/data/db_27001
      indexBuildRetry: true
      journal:
        enabled: true
        commitIntervalMs: 100
      wiredTiger:
        engineConfig:
          cacheSizeGB: 1 #根据内存大小一般给 50%～70%内存
          journalCompressor: "snappy"
          directoryForIndexes: true
        collectionConfig:
          blockCompressor: "snappy"
        indexConfig:
```

```
        prefixCompression: true

    sharding:
      clusterRole: configsvr

    #replication Options
    replication:
       oplogSizeMB: 40960 #40GB
       replSetName: configserver #副本集名称

    #operationProfiling Options
    operationProfiling:
      slowOpThresholdMs: 500
      mode: "slowOp"

    processManagement:

      fork: true
```

config 文件复制到 HE2 和 HE4 机器上：

```
    [work@HE1 etc]$ scp -rp mongodb.conf work@HE2:/home/work/mongodb_sharding/
config_27001/etc/
    [work@HE1 etc]$ scp -rp mongodb.conf work@HE4:/home/work/mongodb_sharding/
config_27001/etc/
```

HE1、HE2、HE4 3 台机器的 config 启动：

```
    [work@HE1 etc]$ numactl --interleave=all /home/work/mongodb_sharding/
3.2.12/bin/mongod                                                    -f
/home/work/mongodb_sharding/config_27001/etc/mongodb.conf

    [work@HE1 ~]$ mongo --host 127.0.0.1 --port 27017
    >config={_id:'shard1',members:[{_id:0,host:'192.168.1.248:27017'},{_id:1
,host:'192.168.1.249:27017'},{_id:2,host:'192.168.1.251:27017'}]}
    >rs.initiate(config)

    [work@HE1 ~]$ mongo --host 127.0.0.1 --port 27018
```

```
    >config={_id:'shard2',members:[{_id:0,host:'192.168.1.248:27018'},{_id:1
,host:'192.168.1.249:27018'},{_id:2,host:'192.168.1.251:27018'}]}
    >rs.initiate(config)

    [work@HE1 ~]$ mongo --host 127.0.0.1 --port 27019
    >
config={_id:'shard3',members:[{_id:0,host:'192.168.1.248:27019'},{_id:1,host
:'192.168.1.249:27019'},{_id:2,host:'192.168.1.251:27019'}]}
    rs.initiate(config)

    [work@HE1 ~]$ mongo --host 127.0.0.1 --port 27001
    >config={_id:'configserver',members:[{_id:0,host:'192.168.1.248:27001'},
{_id:1,host:'192.168.1.249:27001'},{_id:2,host:'192.168.1.251:27001'}]}
    >rs.initiate(config)

    mongos 配置:
    [work@HE1 etc]$ cat mongs.conf
    systemLog:
      destination: file
      path: "/home/work/mongodb_sharding/mongos_27000/log/mongos.log"
      logAppend: true

    #net Options
    net:
      port: 27000
      maxIncomingConnections: 65536
    security:
      keyFile:
/home/work/mongodb_sharding/mongos_27000/etc/mongodb-keyfile-benchmark
      clusterAuthMode: "keyFile"
    sharding:
      configDB:
configserver/192.168.1.248:27001,192.168.1.249:27001,192.168.1.251:27001

    [work@HE1      etc]$/home/work/mongodb_sharding/3.2.12/bin/mongos      -f
/home/work/mongodb_sharding/mongos_27000/etc/mongos.conf&
```

最后可以看到 mongod、config server 和 mongos 都启动成功，如图所示。

```
    [work@HE1 ~]$ mongo --host 127.0.0.1 --port 27000
    mongos>
sh.addShard( "shard1/192.168.1.248:27017,192.168.1.249:27017,192.168.1.251:2
7017")
    mongos> sh.addShard( "shard2/192.168.1.248:27018,192.168.1.249:27018,192.
168.1.251:27018")
    mongos>   sh.addShard(   "shard3/192.168.1.248:27019,192.168.1.249:27019,
192.168.1.251:27019")
```

```
[work@HE1 ~]$ mongo --host 127.0.0.1 --port 27000
Percona Server for MongoDB shell version: 3.2.12-3.2
connecting to: 127.0.0.1:27000/test
mongos> sh.addShard( "shard1/192.168.1.248:27017,192.168.1.249:27017,192.168.1.251:27017")
{ "shardAdded" : "shard1", "ok" : 1 }
mongos> sh.addShard( "shard2/192.168.1.248:27018,192.168.1.249:27018,192.168.1.251:27018")
{ "shardAdded" : "shard2", "ok" : 1 }
mongos> sh.addShard( "shard3/192.168.1.248:27019,192.168.1.249:27019,192.168.1.251:27019")
{ "shardAdded" : "shard3", "ok" : 1 }
```

在 mongos 上创建用户：

```
    >use admin
    >db.createUser(
        {
            user: "sys_admin",
            pwd: "MANAGER",
            roles: [ { role: "root", db: "admin" } ]
        }
      )

    mongos> db.auth('sys_admin','MANAGER')
```

```
mongos> db.createUser(
...         {
...             user: "sys_admin",
...             pwd: "MANAGER",
...             roles: [ { role: "root", db: "admin" } ]
...         }
...     )
Successfully added user: {
        "user" : "sys_admin",
        "roles" : [
                {
                        "role" : "root",
                        "db" : "admin"
                }
        ]
}
mongos> db.auth('sys_admin','MANAGER')
1
mongos> show dbs;
admin   0.000GB
config  0.000GB
```

对 helei 库开启 sharding，能够在状态里看到分片信息、mongos 版本、balancer 是否开启、哪些库开启了分片，以及其主分片是谁：

```
mongos> sh.enableSharding("helei")
mongos> sh.status()
```

```
mongos> sh.enableSharding("helei")
{ "ok" : 1 }
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5a61bb45dcb01eea3201a319")
}
  shards:
        {  "_id" : "shard1",  "host" : "shard1/192.168.1.248:27017,192.168.1.249:27017,192.168.1.251:27017" }
        {  "_id" : "shard2",  "host" : "shard2/192.168.1.248:27018,192.168.1.249:27018,192.168.1.251:27018" }
        {  "_id" : "shard3",  "host" : "shard3/192.168.1.248:27019,192.168.1.249:27019,192.168.1.251:27019" }
  active mongoses:
        "3.2.12-3.2" : 1
  balancer:
        Currently enabled:  yes
        Currently running:  no
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "helei",  "primary" : "shard2",  "partitioned" : true }
```

第 7 章 监控

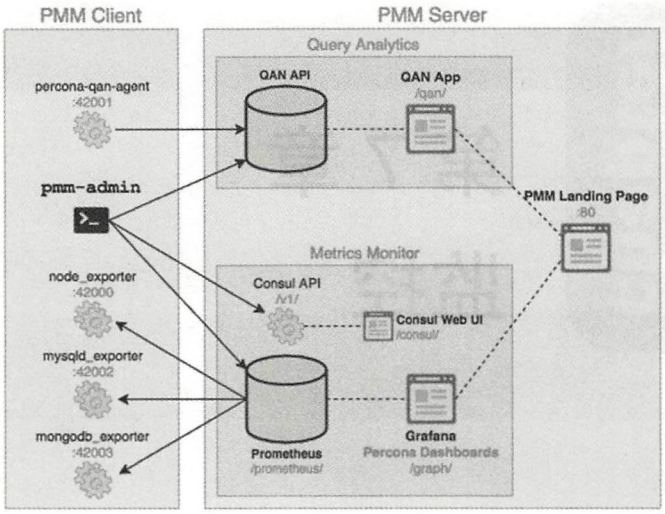
我们已经进入了 MongoDB 数据库最后阶段的学习，一定不要忽视每一个知识点的细节。不积跬步，无以至千里；不积小流，无以成江海。在 MongoDB 数据库知识的体系中怎么能少得了对 MongoDB 监控的学习！监控可以帮助我们排查 MongoDB 的性能瓶颈，方便我们进行调优工作。本章主要学习如何通过部署 PMM 来监控 MongoDB 数据库。

7.1 PMM 监控 MongoDB

PMM 是一款能够监控 MySQL、MongoDB 性能的开源平台，由 Percona 公司开发并提供支持和咨询。

PMM 是免费和开源的解决方案，可以运行在自己的环境中，提供最大的安全性和可靠性，它提供对包括 MongoDB 在内的多种数据库的监控支持。

PMM 架构如下图所示。



PMM Server 作为 Docker 镜像执行 distributed，而 PMM Client 就是一般的 RPM，整个监控需要安装 Server 端和 Client 端。

7.2 Server 组件

Query Analytics(QAN)是用来搜集指令并作性能分析的，其组件分别说明如下。

- QAN API：作为 percona-qan-agent 后端存储和读取 Query 资料用。
- QAN App：提供图形化分析界面。

Metrics Monitor（MM）组件提供了 MySQL 和 MongoDB 历史监控信息，其组件分别说明如下。

- Prometheus：一个开源的服务监控系统和时间序列数据库，它连接到 PMM Client 上的 exporter 以聚集 DB 的监控数据。
- Consul：提供一个 PMM 客户端可以远程列出、添加和删除 Prometheus 主机的 API。
- Grafana：这是一个第三方 Dashboard 和图形构建器，用于可视化 Prometheus 中聚合的数据，以 Web 页面呈现。
- Percona Dashboards：由 Percona 开发的一组用于 Grafana 的仪表板。

7.3 Client 组件

- pmm-admin 是用于管理 PMM Client 的命令行工具。例如，添加和删除要监视的数据库

实例。

- pmm-mysql-queries-0 是一种管理 QAN 代理的服务，从 MySQL 收集查询性能数据并将其发送到 PMM 服务器上的 QAN API。
- pmm-mongodb-queries-0 是一种管理 QAN 代理的服务，从 MongoDB 收集查询性能数据并将其发送到 PMM 服务器上的 QAN API。
- node_exporter Prometheus exporter 用于搜集一般系统信息。
- mysqld_exporter Prometheus exporter 用于搜集 MySQL Server 的信息。
- mongodb_exporter Prometheus exporter 用于搜集 MongoDB server 的信息。
- proxysql_exporter Prometheus exporter 用于搜集 proxysql server 的信息。

PMM Server 官方提供了最简单的 Docker 部署方式，部署起来相当简单。

7.3.1 安装 Docker

```
sudo yum remove docker \
                docker-client \
                docker-client-latest \
                docker-common \
                docker-latest \
                docker-latest-logrotate \
                docker-logrotate \
                docker-selinux \
                docker-engine-selinux \
                docker-engine


sudo yum install -y yum-utils \
  device-mapper-persistent-data \
  lvm2
```

```
sudo yum-config-manager \
    --add-repo \
https://download.docker.com/linux/centos/docker-ce.repo
```

```
sudo yum-config-manager --enable docker-ce-edge
sudo yum-config-manager --enable docker-ce-test

sudo yum install docker-ce

sudo systemctl start docker
sudo systemctl enable docker
```

7.3.2 创建 PMM 数据容器

```
docker pull percona/pmm-server:latest
docker create \
   -v /opt/prometheus/data \
   -v /opt/consul-data \
   -v /var/lib/mysql \
   -v /var/lib/grafana \
   --name pmm-data \
   percona/pmm-server:latest /bin/true
```

此容器不运行，它只是存在，以确保在升级到较新的 pmm-server 时保留所有 PMM 数据。不要删除或重新创建此容器，除非打算清除所有 PMM 数据并重新开始。

选项注释如下。

- docker create：该命令指示 Docker 守护程序从映像创建容器。
- -v：该选项用于初始化数据卷的容器。
- --name：该选项用于引用 Docker 网络中的容器分配一个自定义名称。
- percona/pmm-server:latest ：该选项用于导出容器的镜像名称和版本标签。
- /bin/true ：该选项是容器运行的命令。

7.3.3 运行 PMM 容器，并配置监控登录用户名密码

```
docker run -d \
   -p 80:80 \
   --volumes-from pmm-data \
```

```
    --name pmm-server \
    -e SERVER_USER=admin \
    -e SERVER_PASSWORD=admin \
    --restart always \
    percona/pmm-server:latest
```

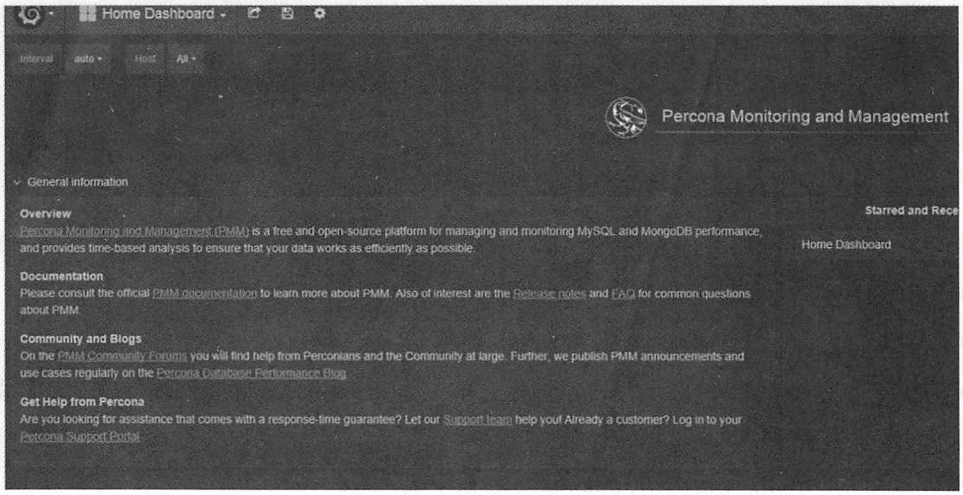
选项注释如下。

- docker run：该命令指示守护程序从镜像运行容器。
- -d：该选项在分离模式（即后台）中启动容器。
- -p：该选项映射用于访问 PMM 服务器 Web UI 的端口。例如-p 8080:80，如果端口 80 不可用，则可以使用登录页面映射到端口 8080。
- --volumes-from：该选项从 pmm-date 容器中装入卷。
- --name：该选项用于引用 Docker 网络中的容器分配一个自定义名称。
- --restart：该选项定义容器的重新启动策略，设置它以 always 确保 Docker 守护程序在启动时启动容器，并在容器退出时重新启动它。
- percona/pmm-server:lates ：是导出容器的镜像名称和版本标签。

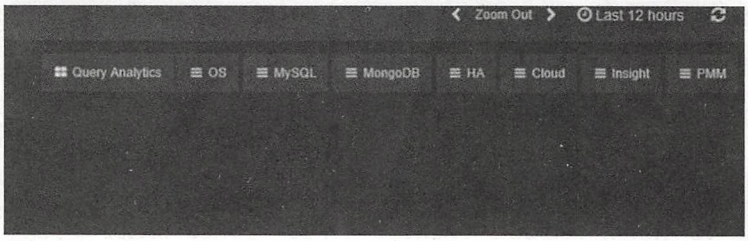
查看 Docker 运行状态。

```
 [root@HE3 -]# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3443717ebc60 percona/pmm-server:latest "/opt/entrypoint.sh" 3 days ago Up
2 days 0.0.0.0:80->80/tcp, 443/tcp pmm-server
d6296ff2a7d3 percona/pmm-server:latest "/bin/true" 3 days ago Created
pmm-data
94d4d2d3aed1 hello-world "/hello" 3 days ago Exited (0) 3 days ago
graciouS-Sha
```

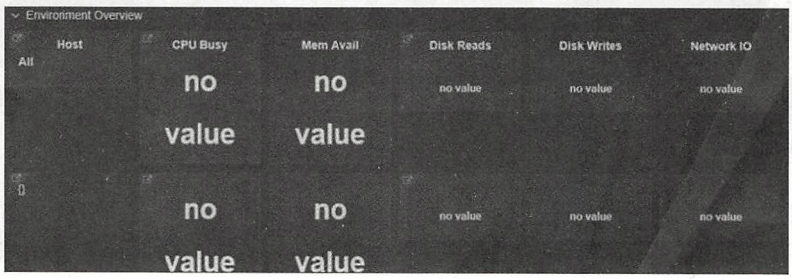
运行 PMM Server 之后，应该可以使用运行容器主机的 IP 地址访问 PMM Web 界面。例如，如果在默认端口 80 上运行 192.168.1.250，则应该可以访问以下内容，如下图所示。



在主页上我们也能够进行历史监控时间的调整，切换 OS 监控、MySQL、MongoDB 监控等，如下图所示。



还可以看到磁盘、CPU、网络 I\O 的一些相关信息，如下图所示。



7.3.4 安装客户端

根据 https://www.percona.com/doc/percona-repo-config/yum-repo.html 配置 yum 源，在需要被监控的 MongoDB 机器上执行如下命令：

```
    sudo yum install http://www.percona.com/downloads/percona-release
/redhat/0.1-4/ percona-release-0.1-4.noarch.rpm
    sudo yum install pmm-client
    pmm-admin config --server 192.168.1.250 --server-user admin
--server-password admin
```

```
[root@HE1 log]# pmm-admin config --server 192.168.1.250 --server-user admin --server-password admin
This client is configured with HTTP basic authentication.
However, PMM server is not.

If you forgot to enable password protection on the server, you may want to do so.

Otherwise, run the following command to reset the config and disable authentication:
pmm-admin config --server 192.168.1.250
[root@HE1 log]# pmm-admin config --server 192.168.1.250 --server-user admin --server-password admin
Unable to connect to PMM server by address: 192.168.1.250

Even though the server is reachable it does not look to be PMM server.
Check if the configured address is correct. %!s(<nil>)
[root@HE1 log]# pmm-admin config --server 192.168.1.250 --server-user admin --server-password admin
OK, PMM server is alive.

PMM Server      | 192.168.1.250 (password-protected)
Client Name     | HE1
Client Address  | 192.168.1.248
```

```
    pmm-admin add mongodb --uri mongodb://sys_admin:MANAGER@127.0.0.1:27017/
admin --cluster=heleitest

    [root@HE1 ~]# pmm-admin add mongodb --uri
mongodb://sys_admin:MANAGER@127.0.0.1:27017/admin --cluster=heleitest
    [linux:metrics] OK, now monitoring this system.
    [mongodb:metrics] OK, now monitoring MongoDB metrics using URI
sys_admin:***@127.0.0.1:27017/admin
    [mongodb:queries] OK, now monitoring MongoDB queries using URI
sys_admin:***@127.0.0.1:27017/admin
    [mongodb:queries] It is required for correct operation that profiling of
monitored MongoDB databases be enabled.
    [mongodb:queries] Note that profiling is not enabled by default because it
may reduce the performance of your MongoDB server.
    [mongodb:queries] For more information read PMM documentation
(https://www.percona.eom/doc/percona-monit:oring-and-managemervt/conf-mongod
b.lvtml).
```

pmm-admin list：用于查看进程状态是否为 running。

```
[root@HE1 etc]# pmm-admin list
pmm-admin 1.8.0

PMM Server      | 192.168.1.250
Client Name     | HE1
Client Address  | 192.168.1.248
Service Manager | unix-systemv

---------------- ----- ----------- -------- ---------------------------------- -------------------
SERVICE TYPE     NAME  LOCAL PORT  RUNNING  DATA SOURCE                        OPTIONS
---------------- ----- ----------- -------- ---------------------------------- -------------------
mongodb:queries  HE1   -           YES      sys_admin:***@127.0.0.1:27017/admin  query_examples=true
linux:metrics    HE1   42000       YES      -
mongodb:metrics  HE1   42003       YES      sys_admin:***@127.0.0.1:27017/admin  cluster=heleitest
```

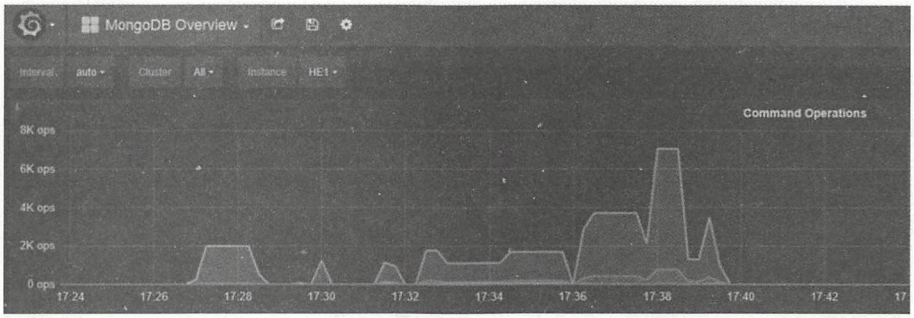
pmm-admin check-network：查看数据和网络是否正常。

```
[root@HE1 ~]# pmm-admin check-network
PMM Network Status

Server Address | 192.168.1.250
Client Address | 192.168.1.248

* System Time
NTP Server (0.pool.ntp.org)         | 2018-03-05 17:53:22 +0800 CST
PMM Server                          | 2018-03-05 09:53:22 +0000 GMT
PMM Client                          | 2018-03-05 17:53:22 +0800 CST
PMM Server Time Drift               | OK
PMM Client Time Drift               | OK
PMM Client to PMM Server Time Drift | OK

* Connection: Client --> Server
------------------- -------
SERVER SERVICE      STATUS
------------------- -------
Consul API          OK
Prometheus API      OK
Query Analytics API OK

Connection duration | 284.911µs
Request duration    | 4.028918ms
Full round trip     | 4.313829ms
```

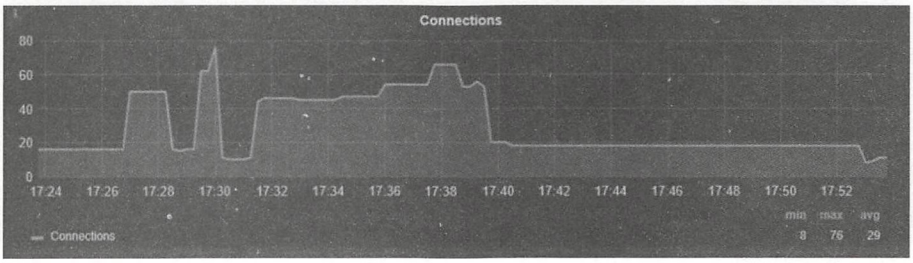
```
* Connection: Client <-- Server
---------------- ----- --------------------- -------- ---------- ----------
SERVICE TYPE     NAME  REMOTE ENDPOINT       STATUS   HTTPS/TLS  PASSWORD
---------------- ----- --------------------- -------- ---------- ----------
linux:metrics    HE1   192.168.1.248:42000   OK       YES        -
mongodb:metrics  HE1   192.168.1.248:42003   OK       YES        -
```

看到 STATUS 是 OK 状态说明正常，如果不是 OK 状态，则检查防火墙、数据库进程和服务器时间是否同步。

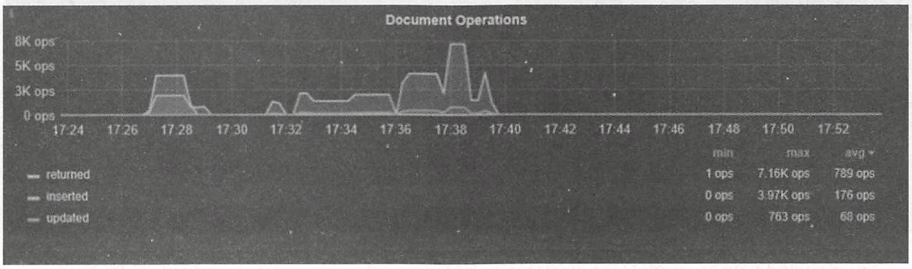
在浏览器中输入我们的 pmm-server 端 IP 地址 192.168.1.250，用户名 admin，密码 admin。进入监控页面，查看 MongoDB 的 QPS，如下图所示。



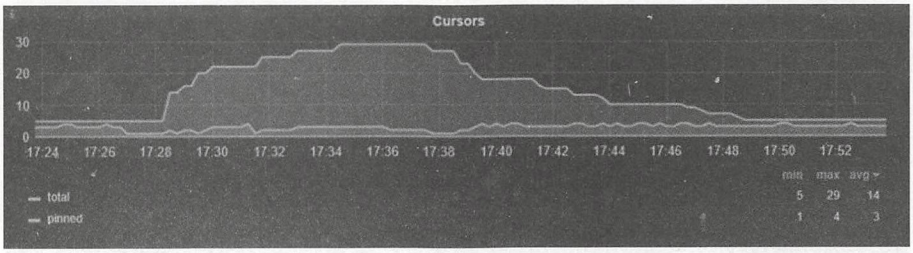
监控连接数情况，如下图所示。



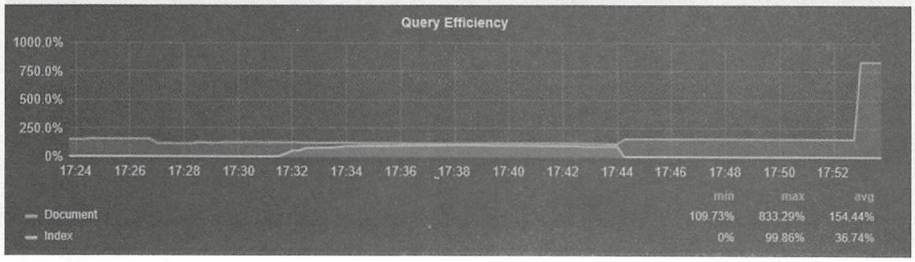
监控文档操作情况，如下图所示。



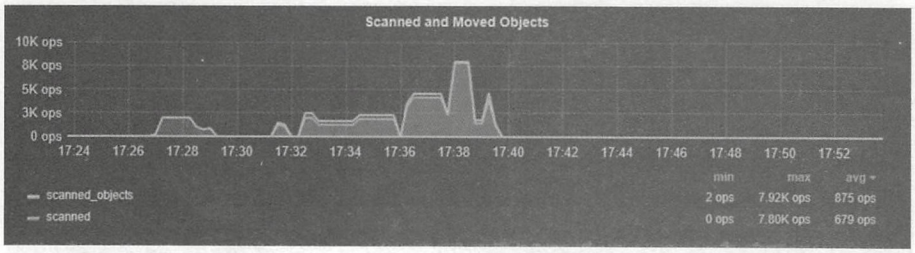
监控 cursor 情况，如下图所示。



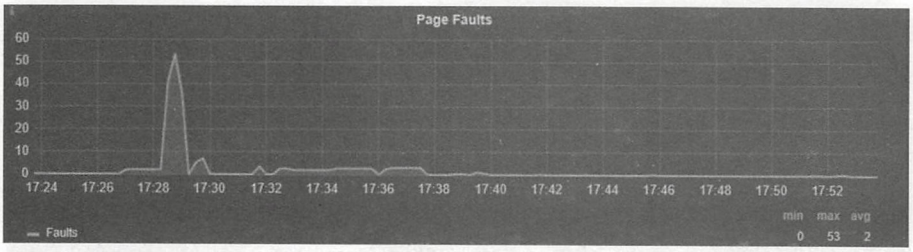
监控查询效率，如下图所示。



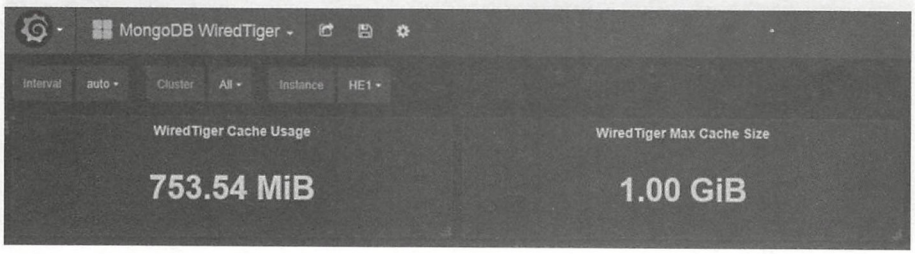
监控扫描和移动的文档数，如下图所示。



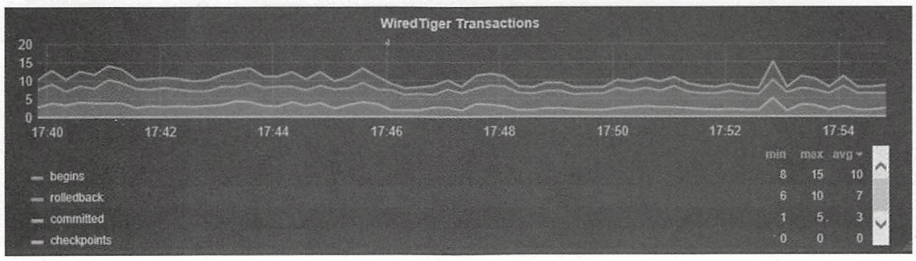
监控 page fault 情况，如下图所示。



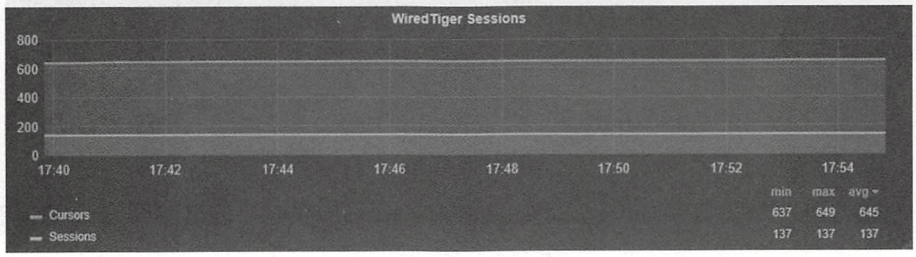
PMM 还可以监控存储引擎的健康程度。例如，监控 WiredTiger 存储引擎的 Cache 使用情况，如下图所示。



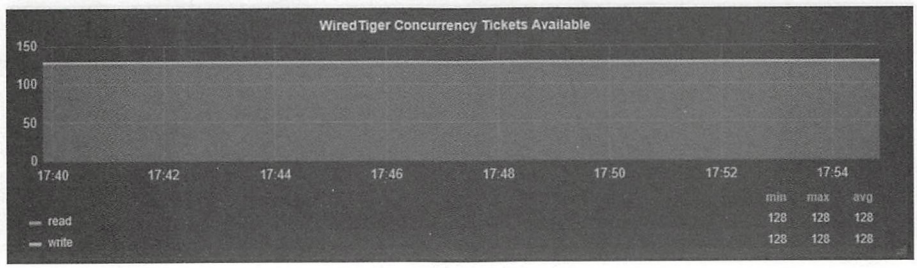
监控 WiredTiger 存储引擎的事务情况，如下图所示。



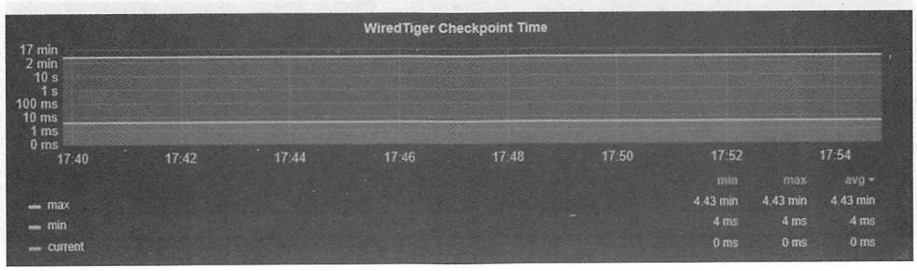
监控 WiredTiger 存储引擎的 Session 情况，如下图所示。



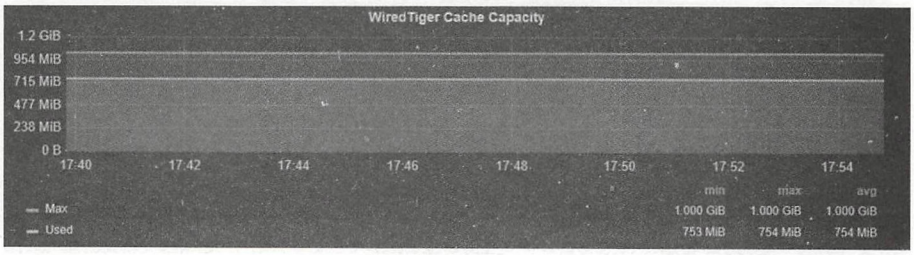
监控 WiredTiger 存储引擎的 Ticket 情况，如下图所示。



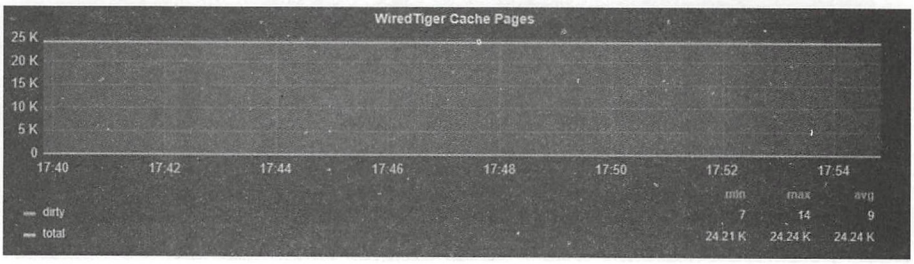
监控 WiredTiger 存储引擎的 checkpoint 情况，如下图所示。



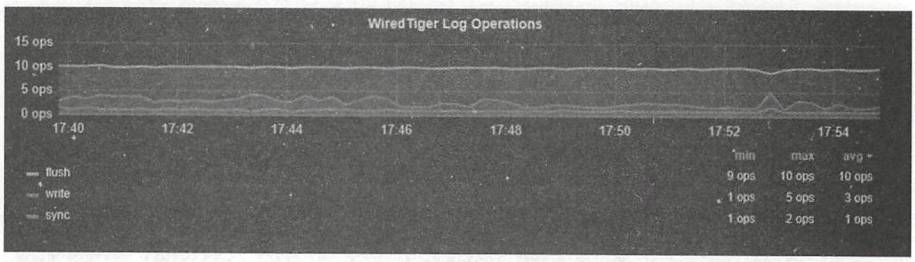
监控 WiredTiger 存储引擎的 Capacity 情况，如下图所示。



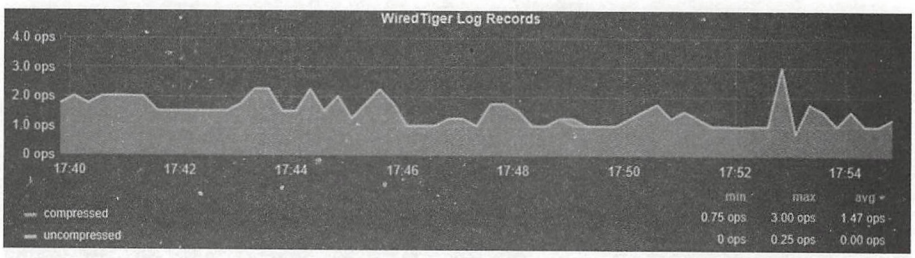
监控 WiredTiger 存储引擎的 Cache Page 情况，如下图所示。



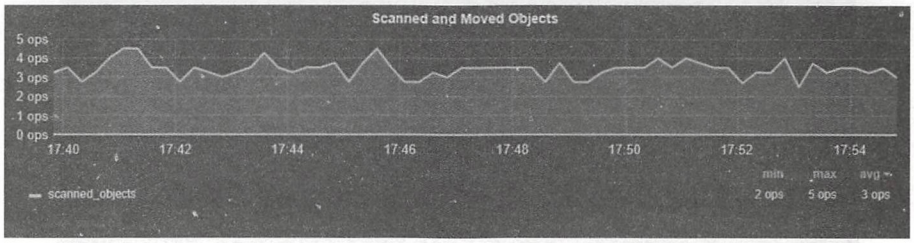
监控 WiredTiger 存储引擎的 log 操作情况，如下图所示。



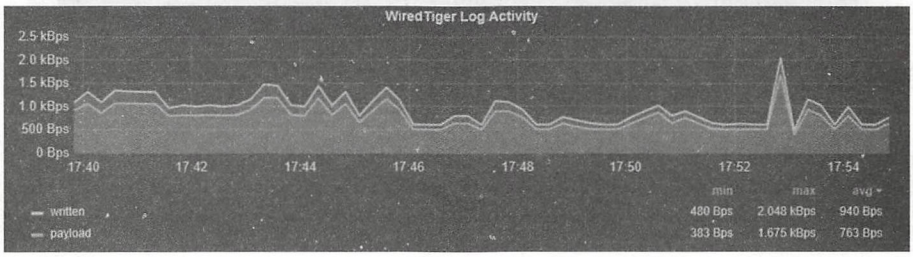
监控 WiredTiger 存储引擎的 log 记录情况，如下图所示。



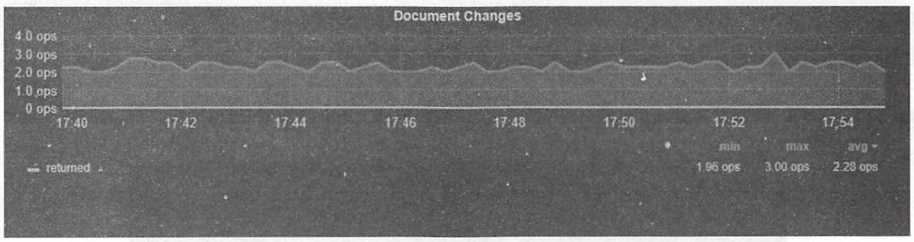
监控 WiredTiger 存储引擎的扫描和移动 Object 的情况，如下图所示。



监控 WiredTiger 存储引擎的日志活动情况，如下图所示。



监控 WiredTiger 存储引擎的文件更改情况，如下图所示。



邮件报警配置如下：

```
    [root@HE3 bin]# docker ps
    CONTAINER ID        IMAGE                          COMMAND                  CREATED
STATUS              PORTS                              NAMES
    3443717ebc60        percona/pmm-server:latest      "/opt/entrypoint.sh"     5
hours ago           Up 2 hours          0.0.0.0:80->80/tcp, 443/tcp   pmm-server
    [root@HE3 bin]# docker exec -it 3443717ebc60 /bin/bash
```

编辑 smtp 如下：

```
    [root@HE3 bin]# vi /etc/grafana/grafana.ini
```

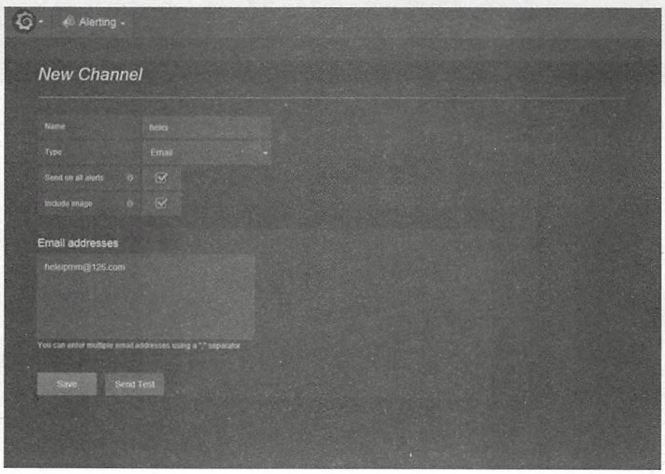
```
#################################### SMTP / Emailing ##########################
[smtp]
enabled = true
host = smtp.126.com:25
user = heleipmm@126.com
# If the password contains # or ; you have to wrap it with trippel quotes. Ex """
"#password;"""
password = helei123456
;cert_file =
;key_file =
skip_verify = false
from_address = heleipmm@126.com
from_name = pmm alert
```

```
#################################### Alerting ###############################
[alerting]
# Disable alerting engine & UI features
enabled = true
# Makes it possible to turn off alert rule execution but alerting UI is visible
execute_alerts = true
```

重启 Docker：

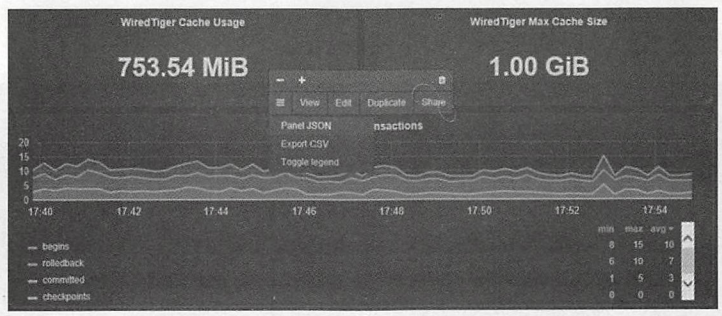
```
 [root@HE3 bin]# docker ps
CONTAINER ID IMAGE COMMAND CREATED
3443717ebc60 percona/pmm-server:latest "/opt/entrypoint.sh" 5 hours ago
[root@HE3 bin]# docker stop 3443717ebc60
3443717ebc60
[root@HE3 bin]# docker start 3443717ebc60
3443717ebc60
```

选择 Alerting 后，填入相应信息，单击 Send Test 按钮，如下图所示。

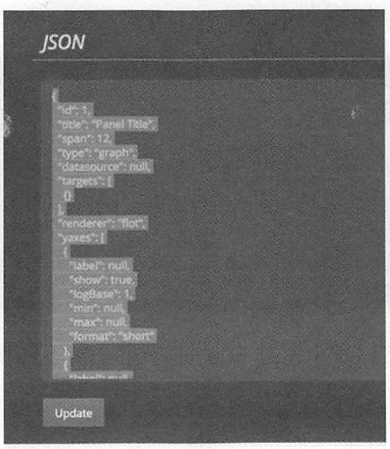


出现下图则表示测试成功。

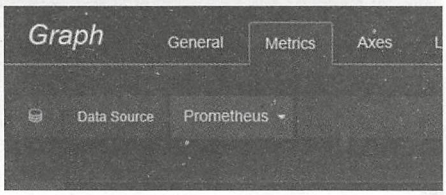
选择一个我们希望产生告警的图，单击鼠标右键，选择其弹出视图左下角的 Panel JSON 选项，如下图所示。



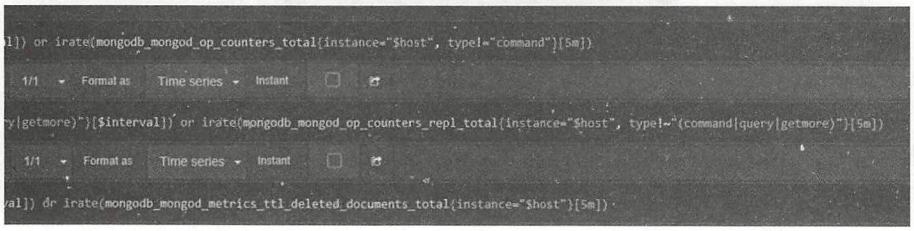
复制 JSON，如下图所示。



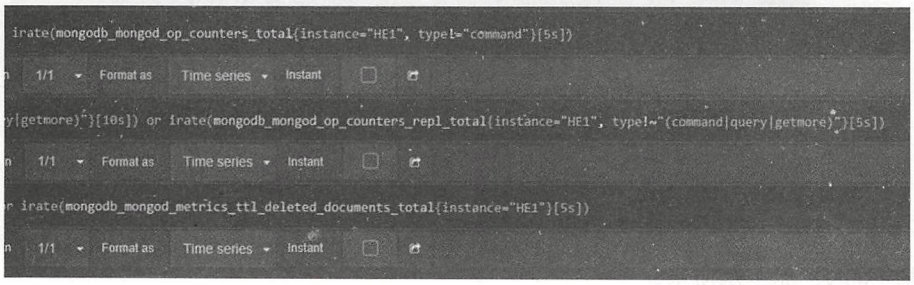
新建一个视图，复制刚刚的 JSON，单击 edit 按钮，选择 Metrics，如下图所示。



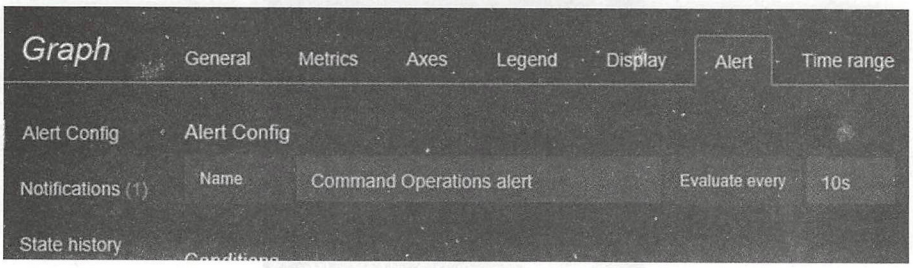
将其中的$描述符都换成具体值。例如，本节中的$host 和$interval，替换为具体的主机名和时间间隔，如下图所示。



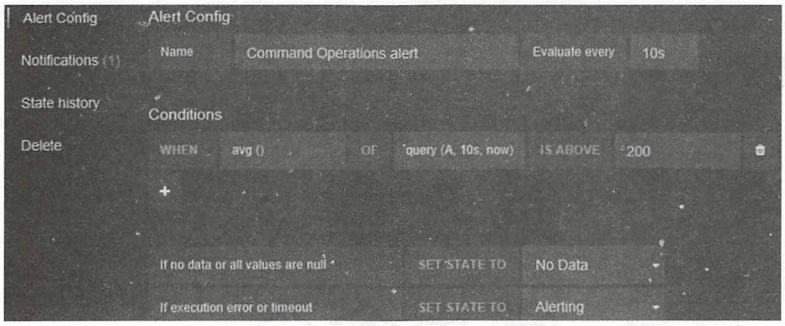
这里的主机名是 HE1，时间是 5s，如下图所示。



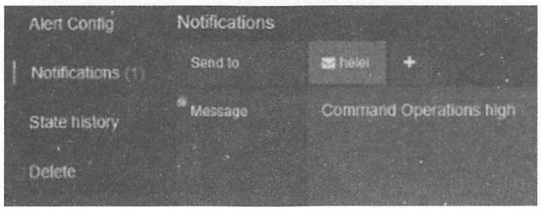
单击 Alert 配置告警阈值，如下图所示。



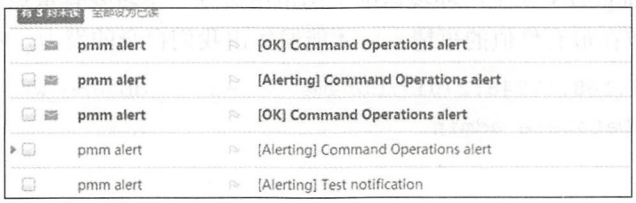
这里是每 10s 检查一次，当 QPS 超过 200 时告警，如下图所示。



单击“通知项”按钮进行内容编辑，这里为了提示 QPS 高于 200，所以写的是 Command Operations high，这个内容是可以自定义的，如下图所示。



稍后邮箱就收到报警信息了，如下图所示。



第8章 常用命令

本章总结了 MongoDB 在生产环境运维中的常用命令。学习要有重点，工作要学会节约时间，把有效的时间放在最有价值的事情上，才能发挥出我们自身的潜力，实现价值的最大化。

```
    mongo    192.168.1.248:27017/dbname    -u    dbname_wr    -p    MANAGER
--authenticationDatabase admin
```

也可以：

```
    ReplSet:PRIMARY> use admin
    ReplSet:PRIMARY> db.auth('helei','MANAGER')

    db.runCommand({"getLastError":1})
    >show dbs: 也可以是 db.adminCommand( { listDatabases: 1 } ), 单位是字节;
    > use admin
    > db.shutdownServer(): 正常停止 MongoDB 数据库进程
    >db.version(): 查看版本号
    >rs.config(): 查看当前复制配置
    >rs.status(): 获取状态 self 表示执行 rs.status 命令的节点, stateStr 表示状态
    >sh.getBalancerState(): 获取 sharding 集群 Balance 是否开启
    >sh.stopBalancer(): 停止 Balancer
    >sh.startBalancer(): 开启 Balancer
```

```
>db.isMaster(): 查看主库信息，主要看 ismaster:true 或者 primary
>sh.disableBalancing("students.grades"): 针对 students 库的 grades 集合关闭 Balance
>sh.enableBalancing("students.grades"): 针对 students 库的 grades 集合开启 Balance
```

配置窗口避免白天操作影响业务，只在凌晨 2 点到 6 点工作：

```
>use config
>db.settings.update(
>   { _id: "balancer" },
>   { $set: { activeWindow : { start : "02", stop : "06" } } },
>   { upsert: true }
>)

>db.serverCmdLineOpts(): 查看当前配置
>db.printReplicationInfo(): 主库查看复制信息
>db.printSlaveReplicationInfo(): 从库查看复制信息、延迟情况
>rs.stepDown(60,5): 让主节点变为备份节点，并在 60s 内无法再次成为主库，等待 5 秒，超时回滚
>rs.freeze(10000): 在备份节点上阻止选举，让其始终在 10000s 内保持备份节点状态，用于维护主节点
>rs.freeze(0): 备份节点被释放，可以在必要时申请为主节点
>db.helei.renameCollection("helei1"): 变更 helei 集合名称为 helei1
>db.currentOp(): 查看当前操作

>use admin
>db.runCommand( { logRotate : 1 } ): 日志切割。

>use admin
>db.fsyncLock(): 锁库
>db.fsyncUnlock(): 解锁，只是解锁请求，可能无法正确解锁，需运行 db.currentOp() 来验证是否还处于锁定状态
>db.getProfilingStatus(): 查看当前 profile 状态
>db.helei.stats(): 查看 helei 集合的信息
>db.currentOP({"ns":"helei"}): 查看当前 helei 库的操作
>db.killOp(): 类似 MySQL kill
>.maxTimeMS(30): 限制命令最大时间为 30 毫秒，Command 后面直接加 maxTimeMS: 45
```

```
    >db.getLastError(): 返回最近一次错误
    > db.serverStatus().opcounters: 查看是否还有流量
```

查看流量：

```
    mongoreplay monitor -i eth0 -e 'port 27020'  --collect json|grep -vE
'("request_data":null|replSet|"ns":"local")'
```

查看流量是否过滤掉 insert 和 update：

```
    ./mongoreplay monitor -i eth0 -e 'port 27020'  --collect json|grep -vE '
("request_data":null|replSet|"ns":"local"|"command":"insert"|"command":"upda
te")'
    > db.serverStatus().opcountersRepl: 查看复制是否还有流量
    >db.serverStatus().globalLock.currentQueue.total: 一直很高，那么存在大量请求
等待锁定的机会，这表示可能会影响性能的并发问题
    >db.serverStatus().extra_info.page_faults: 如果其快速增长则表示物理内存太少，整
个集合访问也会发生效率低下的情况
    >db.serverStatus().globalLock.activeClients: 正在进行或排队的连接总数统计
    >db.serverStatus().connections.current: 连接到当前数据库的客户端数
    >db.serverStatus().connections.available: 未使用连接数
    >db.serverStatus().tcmalloc.tcmalloc.formattedString: 查看相关内存信息
```

更详细的 serverStaus：

```
    db.serverStatus({ "repl":1 })
    db.runCommand({ "serverStatus":1, "repl":1 })
    db.getLogComponents()
    db.setLogLevel(-1, "command" )
    >rs.syncFrom("myHost:27017"): 手动选择复制源，这个操作在 initial sync 期间虽然不
报错，但只有完成 sync 后才会生效，如果想控制复制源，则在 initial sync 前操作
    use admin
    db.runCommand({'resync':1})
```

大于 1 秒查询：

```
    db.currentOp().inprog.forEach(function(item){if(item.ns !="local.oplog.r
s" && item.ns !="" && item.ns !="local.replset.minvalid" && item.secs_running >1)
print(JSON.stringify(item))})
```

在线变更 cache_size：

```
    db.adminCommand({setParameter:      1,      wiredTigerEngineRuntimeConfig:
"cache_size=30G"})
    db.serverStatus().wiredTiger.cache 中的 maximum bytes configured
```

8.1　查询

- DBQuery.shellBatchSize = 50：每页显示多少条；
- db.helei.find().count()：查看表中有多少数据；
- db.helei.findOne()：查看第一条数据；
- db.collection.find().toArray()。

如果你熟悉常规的 SQL 数据，通过下表可以更好地理解 MongoDB 的条件语句查询。

操　作	格　式	范　例	RDBMS 中的类似语句
等于	{<key>:<value>}	db.col.find({"name":"dbapower"}).pretty()	where name = 'dbapower'
小于	{<key>:{$lt:<value>}}	db.col.find({"likes":{$lt:50}}).pretty()	where likes < 50
小于或等于	{<key>:{$lte:<value>}}	db.col.find({"likes":{$lte:50}}).pretty()	where likes <= 50
大于	{<key>:{$gt:<value>}}	db.col.find({"likes":{$gt:50}}).pretty()	where likes > 50
大于或等于	{<key>:{$gte:<value>}}	db.col.find({"likes":{$gte:50}}).pretty()	where likes >= 50
不等于	{<key>:{$ne:<value>}}	db.col.find({"likes":{$ne:50}}).pretty()	where likes != 50

8.2　插入

```
    > for(i=1;i<500;i+=2)db.helei.insert({x:i})
    WriteResult({ "nInserted" : 1 })
```

或者

```
    > for(i=1;i<50000;i++)db.helei.insert({x:i})
    WriteResult({ "nInserted" : 1 })
    db.helei.insert({name:"helei",sex:"man",age:25,time:new Date()})
    db.helei.insert({name:"aixuan",sex:"girl",age:22,time:new Date()})
    db.helei.insert({name:"xiaohong",sex:"girl",age:25,time:new Date()})
```

8.3 修改

```
db.helei.update({"name":"helei"},{$set:{"name":"dbapower"}})
db.helei.update({"name":"helei"},{$set:{"age":"25"}})
db.helei.update({name:"helei"},{"name" : "helei", "sex" : "man", "age" : 26,
"time" : ISODate("2017-03-14T06:08:20.228Z")})  如果不加$set 就成为了替换，即更新为后面列出的文档内容
db.erp_table.renameCollection("erp_t")  修改表名
```

利用 update 增加列：

```
rsBenchmark:PRIMARY> db.helei.update(
{},
{$set:{time:new Date()}},
{multi:true}
)


db.runCommand({ findandmodify  : "helei",
    query: {age: {$gte: 20}},
    sort: {age: -1},
    update: {$set: {name: 'updated'}, $inc: {age: 200}},
    remove: false
});
```

8.4 删除

- db.helei.remove({},1)：删除 helei 表的第一条文档；
- db.helei.remove({"_id":"58b53ee30fd36201476fb62d"})：删除对应 ID 的文档；
- db.helei.drop()：删除表；
- use dbname：切换到待删除库；
- db.dropDatabase()：删除名为 dbname 的库。

分片集群常用命令：

```
>sh.getBalancerState()、sh.isBalancerRunning()：获取 sharding 集群 Balance 是
```

```
否开启;
    >sh.stopBalancer(): 停止 Balancer;
    >sh.startBalancer(): 开启 Balancer, 从 3.4 版本起不会等待 balancing round, 而是
直接启动, 作为 db.adminCommand( { balancerStart: 1 } )的包装;
    >sh.disableBalancing("students.grades"): 针对 students 库 grades 集合关闭
Balance;
    >sh.enableBalancing("students.grades"): 针对 students 库 grades 集合开启
Balance。
```

3.4 版本新命令：

```
    db.adminCommand( { balancerStart: 1 } )、db.adminCommand( { balancerStop:
1 } )、db.adminCommand( { balancerStatus: 1 } )
```

Balance Window 须注意：

（1）平衡器窗口必须足以完成当天插入的所有数据的迁移。

（2）由于数据插入速率可能会根据活动和使用模式而改变，因此确保选择的平衡窗口足以支持部署需求，这一点非常重要。

（3）设置 activeWindow 时不要使用 sh.startBalancer()方法。

取消 Balance Window：

```
    >use config
    >db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

两个参数需要组合起来使用，意思是如果_secondaryThrottle 为 true，则使用 writeConcern 选项来指定迁移时写数据的策略；如果_secondaryThrottle 为 false，则使用{w: 1}选项来指定迁移时写数据的策略。如下命令将 writeConcern 设置为 {w: majority}。从 3.4 版本起，WT 存储引擎默认是_secondaryThrottlefalse。

如果没有设置，则默认使用{w: 2}，要求至少写到目标两个节点（若目标 Shard 是单节点，则退化为{w: 1}）。

```
    >use config
    >db.settings.update(
    >   { "_id" : "balancer" },
    >   { $set : { "_secondaryThrottle" : true ,
    >            "writeConcern": { "w": "majority" } } },
```

```
>   { upsert : true }
>)
```

数据迁移完后，源 Shard 需要将迁移后的 Chunk 移除，默认情况下，源 Shard 会将删除 Chunk 的任务加到一个后台队列，在后台异步删除，然后 Balancer 就可以启动下一次的 Chunk 迁移。用户可以设置_waitForDelete 为true(默认为false)，使源 Shard 在 Chunk 迁移完后同步删除 Chunk 数据：

```
>use config
>db.settings.update(
>   { "_id" : "balancer" },
>   { $set : { "_waitForDelete" : true } },
>   { upsert : true }
>)
```

MognoDB sharding 默认 chunkSize 为 64MB，默认设置在绝大多数场景都是合适的，在某些场景下，用户可能需要修改 chunkSize 配置。如下命令将 chunkSize 修改为 100MB：

```
>use config
>db.settings.save( { _id:"chunksize", value: 100 } )
```

8.5 分片集群常用命令

```
>sh.getBalancerState()sh.isBalancerRunning(): 获取 sharding 集群，Balance 是否开启
>sh.stopBalancer(): 停止 Balancer
>sh.startBalancer() 开启 Balancer，从 3.4 版本起不会等待 balancing round，直接启动。作为 db.adminCommand( { balancerStart: 1 } )的包装
>sh.disableBalancing("students.grades"): 针对 students 库的 grades 集合关闭 Balance
>sh.enableBalancing("students.grades"): 针对 students 库的 grades 集合开启 Balance
>db.locks.find({"state":2}): 没有发现有关 test1 集合相关的锁
>db.runCommand({flushRouterConfig:1}): 没有效果
>db.chunks.find({shard: "rs1","jumbo":true}): 查询哪些是 jumbo chunk
>sh.status(true): 能看到具体是 jumbo 的 chunk
```

```
    db.adminCommand( { balancerStart: 1 } )、db.adminCommand( { balancerStop:
1 } )、db.adminCommand( { balancerStatus: 1 } )
```

Balance window 须注意：

（1）平衡器窗口必须足以完成当天插入的所有数据的迁移。

（2）由于数据插入速率可能会根据活动和使用模式而改变，因此确保选择的平衡窗口足以支持部署需求，这一点非常重要。

（3）设置 activeWindow 时不要使用 sh.startBalancer()方法。

取消 balance window:

```
    >use config
    >db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

两个参数需要组合起来使用，意思是如果_secondaryThrottle 为 true，则使用 writeConcern 选项来指定迁移时写数据的策略；如果_secondaryThrottle 为 false，则使用{w: 1}选项来指定迁移时写数据的策略如下命令将 writeConcern 设置为{w: majority}。从 3.4 版本起，WT 存储引擎默认是_secondaryThrottlefalse，MMAPv1 默认是 true。

如果没有设置，则默认使用{w: 2}，要求至少写到目标两个节点（若目标 shard 是单节点，则退化为{w: 1}）：

```
    >use config
    >db.settings.update(
    >   { "_id" : "balancer" },
    >   { $set : { "_secondaryThrottle" : true ,
    >            "writeConcern": { "w": "majority" } } },
    >   { upsert : true }
    >)
```

数据迁移完后，源 shard 需要将迁移完的 chunk 移除。在默认情况下，源 shard 会将删除 chunk 的任务加到一个后台队列中，在后台异步删除，然后 Balancer 就可以启动下一次的 chunk 迁移。用户可以设置 _waitForDelete 为 true（默认为 false），使源 shard 在 chunk 迁移完后同步删除 chunk 数据：

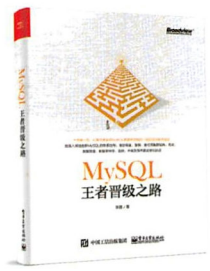
```
    >use config
    >db.settings.update(
    >   { "_id" : "balancer" },
```

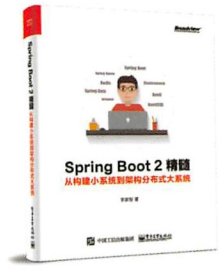
```
>   { $set : { "_waitForDelete" : true } },
>   { upsert : true }
>)
```

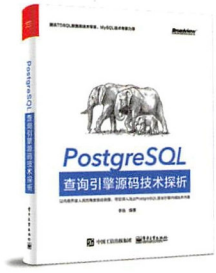
MognoDB sharding 默认 chunkSize 为 64MB，默认设置在绝大多数场景都是合适的，在某些场景下，用户可能需要修改 chunkSize 配置。如下命令将 chunkSize 修改为 100MB：

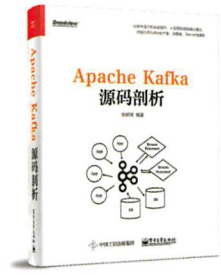
```
>use config
>db.settings.save( { _id:"chunksize", value: 100 } )
```

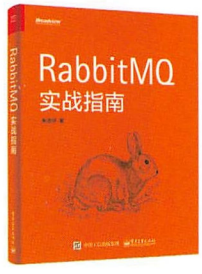
好书分享

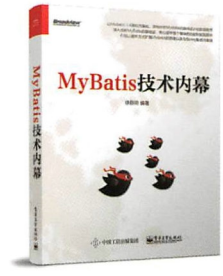












拒绝堆砌臃肿，支持纯正原创

欢迎投稿：chenxm@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

本书汇集了两位作者在MongoDB生产环境实战中遇到的常见问题，包括故障处理、备份恢复、性能调优等，对MongoDB用户来说是一本很好的参考书，可以帮助用户少走弯路，少踩坑。

张友东 阿里云MongoDB专家

我一直以来和很多MongoDB的使用者包括DBA打过不少交道，他们普遍抱怨市面上没有太多可以选择的MongoDB书籍。阅读过《MongoDB运维实战》部分章节后，深感两位作者的专业、细致与用心。大量线上故障实例和部分源码的分析讲解简单明了、通俗易懂。相信本书能对初中级乃至高级DBA在故障定位、线上问题处理及对MongoDB部分原理的理解上都能带来很多帮助。

李丹 MongoDB社区北京分会主席

相同的一个服务，在小规模、大规模乃至海量的情况下，面临的问题和挑战是截然不同的，小规模下的一个可以忽略不计的小问题，在规模放大后，可能会变成一个致命问题。本书的两位作者，基于自己多年在大规模集群使用场景下管理和优化MongoDB服务的实战经验，通过真实的线上案例，以深入浅出的方式，带领读者探索线上问题排查的解决思路与过程，探索如何对大规模集群进行管理和维护，来保障其持续的高可用、高性能。相信读完本书，能让大家对MongoDB服务有一个更深的认识。

李彬 滴滴出行基础平台部高级专家

《MongoDB运维实战》浓缩了两位作者多年MongoDB运维的经验，感谢作者不仅能够快速解决MongoDB线上服务遇到的问题，减少故障损失，还能将其分类整理，帮助更多新老DBA朋友共同进步。其开放、严谨的治学精神值得我为作者点赞。

张良 小米云平台DBA团队负责人

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

责任编辑：陈晓猛
封面设计：李　玲

上架建议：计算机-数据库

ISBN 978-7-121-34989-8

定价：69.00元